LINDA for Case Base Retrieval: A Case for Extending the Functionality of LINDA and its Abstract Machine

D. K. G. Campbell and A. M. Wood
Department of Computer Science, University of York, York YO1 5DD, United Kingdom

H. R. Osborne
University of Huddersfield, Huddersfield HD1 3DH, United Kingdom

D. G. Bridge
Department of Computer Science, University College, Cork, Republic of Ireland

Abstract

Case based reasoning (CBR) retrieval operations with partial ordering similarity measures have been implemented in LINDA to extract generic operations and identify potential parallelism in CBR. Mapping and reduction operations, and constraint matching versions of all retrieval operations were identified, then implemented in LINDA. Their operational semantics were expressed in terms of the CHAM for Liam (the LINDA Abstract Machine) and refined to produce the basis for efficient, parallel implementations of the operations as additional LINDA instructions. The implementation of the constraint matching operations suggests the use of eval() to achieve process migration in distributed systems, and default local tuple space interaction. It as suggested that the constraint matching instructions replace the basic matching retrieval instructions, as they extend the functionality of the matching process by maintaining atomicity of matching and moving the constraint function from the program to the LINDA kernel.

1. Introduction

1.1. LINDA

LINDA [15, 11, 16] is a parallel and distributed coordination language providing communication via tuple spaces (associative memories consisting of bags (or multi-sets) of tuples). Tuples are of two sorts: passive (elements consisting of data only) and active (elements may also consist of processes which on completion replace themselves with their result, and once all processes are complete, form a passive tuple).

The following LINDA instructions operate on tuple spaces:

out(ts,t) takes a (passive) tuple, t, and places it into the tuple space, ts, returning immediately;
in(ts,t) takes a tuple template, t, (containing actual and/or formal arguments), and if it matches a tuple in the tuple space, ts, that tuple is removed and returned to the calling process (with appropriate instantiation of formal arguments). If there is no match available, this will block until one is;
rd(ts,t) behaves like in(ts,t) but returns only a copy of a matching tuple;
eval(ts,t) takes an active tuple, t, and places it into the tuple space, ts, returning immediately. The active tuple can not be retrieved from the tuple space, unlike passive tuples;
tsc() creates a new, uniquely named tuple space, returning the name.

There are also the following predicated instructions, which are not always considered a part of LINDA:

inp(ts,t) behaves as in(ts,t) but does not block, so returns a null value if no match is immediately available;
rdp(ts,t) behaves as rd(ts,t) but, as with inp(), does not block, so returns a null value if no match is immediately available.

Two additional LINDA instructions have been devised and implemented at York. Both of these ‘bulk’ instructions require multiple, named tuple spaces as a matter of course, and are:

collect(ts1,ts2,t) removes all the tuples which, at the moment of invocation, match the template, t, from the source tuple space, ts1, and places them into the destination tuple space, ts2, returning a count of the number of tuples moved. If there are no matches, the count returned is zero, and no tuples are placed in the destination tuple space; [5]
copy-collect(tsi,ts2,t) behaves as collect(tsi,ts2,t) but copies instead of moves the matching tuples. [24]

1.2. Liam

Liam [9] is the L INDA Abstract Machine, constructed using the formalism provided by the Chemical Abstract Machine (CHAM) [2, 4]. The CHAM is based upon a chemical metaphor, it specifies a syntax for molecules, and provides the notion of a sub-solution (chemical solutions can be stratified by encapsulating sub-solutions within membranes that force reactions to occur locally).

A CHAM is specified by defining the following:

- **molecules** $m_1, m_2, \ldots$; terms of algebras (and solutions) with specific operations for the particular CHAM
- **solutions** $S, S', \ldots$; finite multi-sets of molecules $\{m_1, m_2, \ldots, m_k\}$.
- **transformation rules** that determine a transformation relation $S \rightarrow S'$, of which there are two types:
  1. **general laws**, which are applicable to all CHAMs
  2. **specific rules**, which define a given CHAM

Only the general laws involve premises, the specific rules are bound to be elementary rewriting rules.

Molecules exhibiting interaction capabilities are called **ions**, e.g. $\alpha.P$, where the interaction capability of an ion is generally determined only by a part of it called its **valence**, e.g. $\alpha$.

Any solution $S$ can itself be considered as a single molecule, so can appear as a sub-solution of another molecule. This is achieved by the **membrane operator** $\{\}[]$, which transforms a solution into a single molecule which is contained within a membrane, where it can evolve on its own, obeying the same rules as the global solution but in isolation. However, to realise global communication, it is required that the membranes be porous to valences, hence the **airlock constructor** $\triangleleft$.

An airlock is a molecule $m \triangleleft S$, built from a molecule $m$ and a solution $S$. The airlock creation mechanism extracts any molecule from a solution, and puts the rest of the solution within a membrane. So, the airlock law can be used to isolate a molecule from one level of solution, and enable it to react with a molecule of the surrounding solution.

By creating or removing airlocks, a solution can propose several valences in succession to its environment until a communication takes place.

There are four general laws, which specify how reactions defined by specific transformation rules can take place, and how membranes and airlocks behave:

1. **Reaction Law**: an instance on the right hand side of a rule can replace the left hand side.
2. **Chemical Law**: reactions can be performed freely within any solution

   $$S \rightarrow S'$$

   $$(S \uplus S'') \rightarrow S' \uplus S''$$

   ($S \uplus S'$ is the multi-set union of $S$ and $S'$.)
3. **Membrane Law**: a sub-solution can evolve in any context

   $$S \rightarrow S'$$

   $$(C[S])[] \rightarrow C[S'][]$$

   ($C[ ]$ denotes a molecule with hole [ ] in which to place another molecule.)
4. **Airlock Law**

   $$\{m\} \uplus S \leftrightarrow \{m \triangleleft S\}$$

   (The double arrow means that this law is reversible.)

The chemical and membrane laws are the only laws to involve premises, and factor out what are usually called structural rules in particular calculi.

Specific rules are of the form: $m_1 \ldots m_k \rightarrow m_1' \ldots m_k'$. They are presented as rule schemata, where the actual rules are instances of those schemata. To avoid multi-set matching, the sub-solutions in specific rules are either a single solution variable $S$ that generates all solutions, or $\{m\}$ where $m$ is a single molecule schema.

There are three kinds of rule (though the distinction is not enforced by the formalism):

1. **heating** $(S \rightarrow S')$, which breaks non-ion molecules into sub-molecules;
2. **cooling** $(S \rightarrow S')$, which combines molecules into a complex molecule
3. **reaction** $(S \rightarrow S')$, which irreversibly changes the information in the solution, usually involving ions (molecules which can not be heated further)

The heating and cooling rules are often written together as $\rightarrow$, which signifies structural manipulation.

The CHAM is intrinsically parallel, with mutually exclusive reactions being able to occur in parallel. When the sets of possible reactions are not mutually exclusive, a reaction is chosen non-deterministically. Also, since multi-sets are intrinsically unordered, the chemical concurrency is naturally associative and commutative.

Liam is an extension of the CHAM for L INDA in [12]. (See Figure 1 where $op$ indicates either $rd$ or $in$, $\perp$ denotes a formal parameter, $\#t$ is the arity of tuple $t$, $t_i$ is the $i^{th}$
its most extreme form, only compute case similarity during DAG, where proximity in the data structure denotes similarity. In this approach, similar cases are retrieved by transforming \(\sigma\) to a disjunctive normal form — i.e. \(\sigma = \sigma_1 \lor \sigma_2 \lor \cdots \lor \sigma_n\). The maxima (the cases returned) will be constructed from simpler, atomic similarity measures. In [21] it is shown that cases can be retrieved by transforming \(\sigma\) to a disjunctive normal form — i.e. \(\sigma = \sigma_1 \lor \sigma_2 \lor \cdots \lor \sigma_n\). The maxima (the cases returned)
are then rewritten as:
\[
\prod_{1 \leq i \leq n} \prod_{(s_i, k)} C.
\]

This obviously presents opportunities for parallelisation, as each set of maxima \(\prod_{(s_i, k)} C\) can be computed in parallel.

By modelling cases as tuples — the basic constructor in L INDA — case bases and the algorithms involved in case base retrieval can be realised in L INDA. Given that as information systems increase the size of their information bases, they will be required to be distributed in order to acquire the necessary memory and computational resources. By using L INDA, case bases can be distributed over a number of workstations, so allowing the modelling of distributed case bases.

1.4. Overview

The aim of this paper is to present the analysis of an investigation into the identification of generic operations for case base retrieval in L INDA; this expands upon the work presented in [10].

In section 2, the generic operations for reduction, mapping and constraint matching are identified and their implementations in L INDA and specifications in the CHAM for Liam are presented. Finally, the conclusions are drawn in section 3.

2. Generic operations for CBR in L INDA

The goal of this work was to identify generic operations for CBR retrieval in L INDA, with the intention of implementing those operations efficiently so as to increase performance.

This process was initially seen as unlikely to produce radically new generic operations. Instead, it was regarded as solely identifying which in the range of general operations are used in this application. That is, identifying instances of algorithmic skeletons [13, 10], where an algorithmic skeleton is a high-level, parallel programming language construct encapsulating the expression of parallelism, communication, synchronisation and embedding.

The generic operations identified are detailed in the following sub-sections.

2.1. reduce()

2.1.1. Identifying reduction. From equation (1), the maxima of the cases can be expressed as:
\[
\prod_{1 \leq i \leq n} \prod_{(s_i, k)} C
\]

which is a reduction (see [3]) applying an intersection over the \(n\) individual sets. That is to say, equation (1) can be expressed as:
\[
\tau_2 / \{\prod_{(s_i, k)} C, \prod_{(t_2, k)} C, \ldots, \prod_{(s_n, k)} C\}
\]

\[
= \prod_{(s_i, k)} C \cap \tau_2 \prod_{(t_2, k)} C \cap \ldots \cap \tau_2 \prod_{(s_n, k)} C
\]

where \(\cap \) is the commutative and associative operation for intersecting a pair of sets, and \(/\) is the reduction operator.

The maxima of a set \(C\), given a partial ordering \(\subseteq\), are:
\[
\prod_{\subseteq} C = \{x \in C \mid \forall y \in C : x \subseteq y \Rightarrow y \subseteq x\}
\]

\[
= \bigcap_{y \in C} \{x \in C \mid x \subseteq y \Rightarrow y \subseteq x\}
\]

(2)

Therefore, combining equations (1) and (2), case base retrieval can be represented as a pair of nested reductions applying intersection:
\[
\prod_{(s_i, k)} C = \bigcap_{1 \leq i \leq n} \bigcap_{y \in C} \{x \in C \mid x(s_i, k)y \Rightarrow y(s_i, k)x\}
\]

(3)

2.1.2 Reduction in L INDA. Reduction has been implemented in L INDA as the reduce() operation, details of which can be found in [6], where reduce() is referred to as closure().

As an overview, the reduce() function is given an operation and a tuple space containing tuples. Each tuple in the given tuple space contains the name of a tuple space upon which the operation is to be performed.

Pairs of tuples are then consumed (sequentially or in parallel) from the source tuple space, the operation is applied to the tuple spaces identified by their tuple space names, returning a new tuple space name, identifying the resultant tuple space, which is placed in a tuple in the source tuple space.

This is then repeated until a single tuple remains in the source tuple space. That tuple contains the name of the tuple space containing the resulting reduction, and that name is returned.

2.1.3 Reduction in the CHAM. The specification of the operational semantics for reduce() is also presented in [6], where reduce() is specified in terms of the CHAM for Liam. This represents a specification of reduce() based upon its implementation in terms of L INDA instructions, and is illustrated in Figure 2 for the sequential case (the parallel case being too large to include, consisting of twelve long rules, each calling rules to interpret the L INDA instructions).

Besides simply specifying the semantics for the L INDA implementation of reduce(), the specification was then
Definition: CHAM \[ reduce() \]

reduce\( (t\cdot s, f, t\cdot s) \cdot P \rightarrow \) 
\( \text{tsc}(\text{tmp}) \cdot \text{copy}(t \cdot s, \text{tmp}, (\perp: t\cdot sname), \perp) \cdot \text{reduce}'(\text{tmp}, f, t\cdot s) \cdot P \)

reduce\( ' \)\( (t\cdot s, f, t\cdot s) \cdot P \rightarrow \) 
\( \text{in}(t\cdot s, 1) \cdot \text{inp}(t\cdot s, 2) \cdot \text{reduce}''(t\cdot s, 2, t\cdot s, f, t\cdot s) \cdot P \) 
where \( t\cdot s1 = (\perp: t\cdot sname) \) 
\( t\cdot s2 = (\perp: t\cdot sname) \)

reduce\( '' \)\( (t\cdot s1, \text{NULL}, t\cdot s, f, t\cdot s) \cdot P \rightarrow \) 
\( P[t\cdot s1/t\cdot s] \)

reduce\( ''' \)\( (t\cdot s1, t\cdot s2, t\cdot s, f, t\cdot s) \cdot P \rightarrow \) 
\( f(t\cdot s1, t\cdot s2, t\cdot s3) \cdot \text{out}(t\cdot s3) \cdot \text{reduce}''(t\cdot s, f, t\cdot s) \cdot P \) 
where \( t\cdot s2 \neq \text{NULL} \)

Figure 2. The CHAM for the reduce() operation in Liam

Definition: CHAM \[ \text{refined parallel reduce()} \]

reduce\( (t \cdot s : R, (n) : \| \text{reduce}(t \cdot s, f, t\cdot s) \cdot P \triangleleft S) \rightarrow \) 
\( (t \cdot s : R, (n) : \| \text{tsc}(t\cdot s') \cdot \text{reduce}''(t\cdot s', f, t\cdot s) \cdot P \triangleleft S) , (t\cdot s') : R \)

reduce\( ' \)\( (t\cdot s') : \{ (t\cdot s1 : t\cdot sname) \triangleleft \} \| \} , (n) : \| \text{reduce}'(t\cdot s', f, t\cdot s) \cdot P \triangleleft S) \rightarrow \) 
\( (n) : \| P[t\cdot s1/t\cdot s] \triangleleft S \)

reduce\( ' ' \)\( (t\cdot s') : \{ (t\cdot s1 : t\cdot sname) \triangleleft \} \{ (t\cdot s2 : t\cdot sname) \triangleleft R) \} , (n) : \| \text{reduce}'(t\cdot s', f, t\cdot s) \cdot P \triangleleft S \} \rightarrow \) 
\( (t\cdot s') : \{ f(t\cdot s1, t\cdot s2) : t\cdot sname \triangleleft R) \} , (n) : \| \text{reduce}''(t\cdot s', f, t\cdot s) \cdot P \triangleleft S \} \)

Figure 3. The refined CHAM for the parallel reduce() operation in Liam

refined to a new CHAM, preserving the semantics whilst reducing the numbers of rules.

This is presented in Figure 3, which is the parallelised version, consisting of only three rules (of which only one calls a Linda instruction rule), compared to the twelve rules (each calling Linda instruction rules) in the unrefined specification. This CHAM provides the basis for an efficient \( O(n) \) speedup on the unrefined version and parallel implementation of reduce() as a primitive instruction for Linda, as it can no longer be implemented in terms of Linda instructions.

2.1.4 Other applications. The reduce() operation is a basic operation over bags [3], and can be used in any situation where a commutative and associative operation is to be applied to a collection of tuple spaces to produce a single resultant tuple space. Similarly, reduce() can also be described as an instance of the Iterative Transformation skeleton (see [10]).

2.2. map()

2.2.1 Identifying and implementing mapping. As is illustrated in [8], the intersection of a pair of tuple spaces can be represented by a mapping operation. An algorithm for finding the intersection of a pair of tuple spaces is shown in Figure 4. This is written in a pseudo-C-Linda syntax.

The algorithm takes two tuple spaces, \( t\cdot s1 \) and \( t\cdot s2 \) of type \( t\cdot sname \), and a tuple template \( \text{template} \). It then produces copies of the given tuple spaces, so ensuring that they are static, i.e., they will not be changed by other processes during the course of this process running. The algorithm then repeatedly consumes a tuple, instantiated as \( \text{tup} \), from the static copy of tuple space \( t\cdot s1 \), collecting any matching ones from the static copy of the other tuple space \( t\cdot s2 \) and placing them in the destination tuple space \( t\cdot s0 \). Each loop iteration consumes a different tuple used as a template by the collect(), hence interference between concurrently operating collect() instructions will not occur. Therefore, this loop body can be performed in parallel (given a suitable means of implementing parallel instances of each iteration of the for loop). Finally, the name of the new tuple space containing the intersection of the pair of given tuple spaces is returned.

Considering the structure of the intersect2() algorithm, it basically consumes a tuple from a tuple space, then uses that tuple as argument to a collect() instruction. However, if one were to generalise this by replacing the collect() with any given function, one would have an algorithm which applied the given function to each tuple in a tuple space. This generalised operation has been named map(), due to its similarity to mapping in functional languages, and is shown in Figure 5. The map() operation takes a tuple space \( t\cdot s1 \), a function \( \text{fun} \) (which is effectively a \( \lambda \)-expression), and a template \( \text{template} \). A copy is taken of the tuple space to ensure that it remains static, and to facilitate the retrieval of each tuple uniquely. Then tuples are repeatedly consumed, instantiated as \( \text{tup} \), and operated on by the supplied function, which places its results in a destination tuple space.

The intersect2() operation may now be implemented as in Figure 6.
2.2.2 Mapping in the CHAM. The operational semantics of \texttt{map()} are presented in [9], where they are specified in terms of the CHAM for Liam. This is illustrated in Figure 7 and represents a specification of the sequential implementation of the operation based upon its implementation in terms of the LINDA instructions.

The specification in the CHAM provides the basis for an implementation of \texttt{map()} as an instruction for LINDA given a small amount of refinement of the CHAM specification.

2.2.3 Other applications. Mapping is a basic operation over bags [3], and is ubiquitous throughout computer programming, so this new operation is suitable for many other situations beside CBR. Similarly, the \texttt{map()} operation can be seen as an instance of the \textit{Task Queue} skeleton (see [10]).

2.3. Constraint matching

2.3.1 Identifying constraint matching. Constraint matching is not so much a generic operation in the style
Definition: CHAM \[\text{map}(\cdot)\]

\[
\text{map}(ts,a,f).P \rightarrow \text{tsc}(\text{tmp}).\text{copy}(ts,\text{tmp},a,c).\text{map}'(c,\text{tmp},a,f).P
\]

\[
\text{map}'(n):\{\text{map}'(0,\text{tmp},a,f).P \triangleq S\},(\text{tmp}):\{\}\} \rightarrow (n):\{P \triangleq S\}
\]

\[
\text{map}'(c,\text{tmp},a,f).P \rightarrow \text{map}'(c-1,\text{tmp},a,f.in(\text{tmp},a).f(a).P)
\]

Figure 7. \text{map}(\cdot) in Liam

The simplest way to implement constraint matching is, in general, to repeatedly perform the non-constraint matching operation and test whether retrieved tuples satisfy the given constraint. This is illustrated in Figure 8 for the \text{ccopy-collect()} operation in a pseudo-C-LINDA style, where the operation is given the two tuple spaces \text{t}1 and \text{t}2, the tuple template \(t\), and the constraint function \(f\).

This is clearly a wasteful option, particularly on a distributed implementation of LINDA, where the tuple space being operated upon may be stored on a different processor to the one on which the process is executing. In such a case a significant amount of time would be wasted in the communication overhead of transporting tuples across the network that it was then found did not fit the matching criteria.

Alternatively, the \text{eval()} instruction, which has hitherto tended to be neglected amongst the LINDA instructions, can be employed to place the testing part of the constraint matching operation in the tuple space being operated upon and on the processor upon which it is stored, i.e., migrating the processing to the data in a distributed environment.

The procedure in this case is to first create a temporary tuple space then to use \text{eval()} to spawn a subsidiary process in the tuple space being operated upon. The process then waits for the active tuple to be replaced by a passive tuple to signal that the search is complete before it consumes the suitably matched tuple from the temporary tuple space. The tuple is then returned as the result. This is illustrated in Figure 9 for the \text{ccopy-collect()} operation, which differs slightly from the above description in that for bulk operations the count of matched tuples is returned. In this figure the structures terminated by angled brackets are tuples.

The subsidiary process working in the tuple space being operated upon performs the repeated testing of tuples (locally) for satisfaction of the given constraint. Once a suitable tuple is matched, it is placed in the temporary tuple space, and the process returns a flag, so causing the active tuple to be replaced by a passive one. This is illustrated in Figure 10.

This distribution of processing is akin to the view of distributed Intelligent Knowledge Manipulation Systems in the future performing parts of a given query at distant sites containing the relevant data, thus taking the processing to the data node, rather than passing large amounts of data across the network.

2.3.2 Constraint matching in LINDA. Constraint matching has been implemented for all of the LINDA retrieval instructions in \[7\]. Each instruction, \text{in()}, \text{rd()}, \text{inp()}, \text{rdp()}, \text{collect()} and \text{copy-collect()} has been generalised to match tuples on any given constraint function, and is named \text{cin()}, \text{crd()}, \text{cinp()}, \text{crdp()}, \text{ccollect()} and \text{ccopy-collect()} respectively.

of an algorithmic skeleton but a generalisation of existing retrieval instructions. It extends the functionality of each retrieval instruction by enhancing the matching process to any given constraint function. This is considerably different from supplementing the basic instructions with algorithmic skeletons.

In constraint matching the basic matching algorithm is replaced by a given constraint function. This function specifies the constraint on a tuple for it to match.

The requirement for constraint matching in CBR is in finding the maxima of a set according to a partial ordering given by an atomic similarity measure, i.e., from equation (1):

\[
\cap_{(\sigma, k)} C
\]

which is rewritten from equation (2) as:

\[
\cap_{(\sigma, k)} C = \bigcap_{y \in C} \{ x \in C \mid x(\sigma_k) y \Rightarrow y(\sigma_k) x \} \quad (4)
\]

If \(C\) contains \(n\) tuples, and \(C'\) is a copy of \(C\), the calculation of the subsets to be intersected in equation (4) can be realised with the following piece of pseudo-C-LINDA code:

```pseudo-c
for(j=0;j<n;j++){
    y=in(C',template);
    ccopy-collect(C,R[j],S(i,y));
}
```

where \(R[j]\) is the result tuple space containing the maxima according to the partial ordering according to the given \(y\), and \(S(i,y)\) is the partial ordering \(x(\sigma_k) y \Rightarrow y(\sigma_k) x\), where \(x\) is the potentially matching tuple being tested in \(C\) by \text{ccopy-collect()}. Additionally, \text{ccopy-collect()} is the constraint matching version of \text{copy-collect()}.

So, this can be read as: for each \(y\) in \(C'\), place in \(R[j]\) those tuples satisfying \(S(i,y)\) from \(C\).
```c
int ccopy-collect(tsname ts1, tsname ts2, tuple t, (*f)(void*)) {
    ts3 = tsc();
    c = 0;
    n = ccopy-collect(ts1, ts3, t);
    for (i = 0; i < n; i++) {
        tt = in(ts3, t);
        if ((*f)(*tt)) {
            c++;
            out(ts2, tt);
        }
    }
    return (c);
}
```

**Figure 8. Simulating ccopy-collect() at the user process**

```c
int ccopy-collect(tsname ts1, tsname ts2, tuple t, (*f)(void*)) {
    ts3 = tsc();
    eval(ts, <ccopy-collect1(t, f, ts1, ts3)>);
    in(ts1, <FLAG>);
    return (collect(ts3, ts2, t));
}
```

**Figure 9. Emulating ccopy-collect() using eval()**

```c
void ccopy-collect1(tuple t, (*f)(void*), tsname ts1, tsname ts3) {
    ts4 = tsc();
    n = ccopy-collect(ts1, ts4, t);
    for (i = 0; i < n; i++) {
        tt = in(ts4, t);
        if ((*f)(*tt))
            out(ts3, tt);
    }
    return (FLAG);
}
```

**Figure 10. The active tuple process for ccopy-collect()**

Using the `eval()` instruction in this way suggests the use of default, local tuple spaces. So, each process could by default interact with whatever tuple space it is located in instead of solely by naming the tuple space for each interaction.

### 2.3.3 Constraint matching in the CHAM.

The operational semantics of the constraint matching versions of each of the retrieval instructions have been presented in [7] in terms of the CHAM for Liam. This was achieved by generalising the CHAM rules for the basic matching versions of the retrieval instructions. This took the form of the retrieval instruction accepting an additional parameter indicating the constraint function. The constraint function parameter served as a tag to indicate which set of CHAM rules would be applied to perform the matching. Therefore, each constraint function is specified by its own set of CHAM rules. So, any constraint that can be specified in terms of a CHAM can be supplied as the constraint to a constraint matching retrieval operation. This resulted in effectively a new Liam, replicating the old version and replacing the instructions with constraint matching versions.

This is illustrated in Figure 11 for the `cin()` and `crd()` instructions without multiple tuple spaces, and should be compared with the corresponding rules in the CHAM for the basic Linda in Figure 1.

The specification of the constraint matching operations in terms of the CHAM suggests a means for implementing the constraint matching retrieval operations as dedicated Linda instructions.

In [10] the modelling of the performance for a simple implementation of the constraint matching operations in terms of Linda is compared with the low-level implementation from the derived CHAM specification. These results show a hundred-fold improvement in performance for the low-level implementation on the particular platform (a SPARCstation 5 connected to a Transtech Paramid). Therefore, the low-level specification in terms of the CHAM for Liam has
Definition: CHAM [CMLINDA match]

- \(\forall \text{pair} \) \( \text{op}^p_j(\langle i, s_i \rangle, \langle i, t_i \rangle) \mid 1 \leq i \leq \#s = \#t \) \( P \leftrightarrow \text{op}(s, f).P, t \)
- \(\forall \text{match} \) \( \text{op}^\text{match}_1(\langle i, a : \tau \rangle, \langle i, a : \tau \rangle) \leftrightarrow \text{op}(\langle i, a : \tau/a : \tau, \text{match} \rangle, \langle i, a : \tau \rangle) \)
- \(\forall \text{match} \) \( \text{op}^\text{match}_2(\langle i, a : \tau \rangle, \langle i, a : \tau \rangle) \leftrightarrow \text{op}(\langle i, a : \tau/a : \tau, \text{match} \rangle, \langle i, a : \tau \rangle) \)
- \(\forall \text{match} \) \( \text{op}^\text{match}_3(\langle i, a : \tau \rangle, \langle i, a : \tau \rangle) \leftrightarrow \text{op}(\langle i, a : \tau/a : \tau, \text{match} \rangle, \langle i, a : \tau \rangle) \)
- \(\forall \text{crd} \) \( \text{crd}(\langle i, r_i/s_i, f \rangle, \langle i, t_i \rangle) \mid 1 \leq i \leq \#t = \#r = \#f \) \( P \leftrightarrow \text{P[r/s], t} \)
- \(\forall \text{cin} \) \( \text{cin}(\langle i, r_i/s_i, f \rangle, \langle i, t_i \rangle) \mid 1 \leq i \leq \#t = \#r = \#f \) \( P \leftrightarrow \text{P[r/s]} \)

Figure 11. CHAM for constraint matching LINDA

again produced the basis for a more efficient implementation compared to a simple specification of the LINDA-level implementation.

As the constraint matching operations are generalisations of the basic retrieval instructions, it can be argued that they replace them. This is due to the basic retrieval instructions being only a special case of the constraint matching versions, and the basic instructions can be trivially specified in terms of the CHAM for the constraint matching operations.

2.3.4 Other applications. Free Flight [25] can be considered as an application area for constraint matching in LINDA, where an aircraft would request the location tuples of aircraft that are within a certain radius of itself. These aircraft location tuples are stored in a distributed, open system including aircraft and ground-stations.

3. Conclusions

LINDA was chosen to implement case based reasoning, partly due to its tuples and tuple spaces being ideal representations for cases and case bases, respectively. The goal of the research was to identify generic operations for this application with the aim of implementing them more efficiently.

The basic generic operations identified were \(\text{reduce}()\) and \(\text{map}()\), both of which are basic operations over bags.

The operations have been implemented in terms of the basic LINDA instructions, and they have been specified in terms of the CHAM for the LINDA Abstract Machine (Liam). The specification of \(\text{reduce}()\) has been refined to an efficient and parallel form with \(O(n)\) speed-up over the unrefined version, so providing the basis for an implementation of \(\text{reduce}()\) as a primitive instruction for LINDA and improving the performance of case base retrieval. Similarly, the specification of \(\text{map}()\) has been refined and provides the basis for an implementation of \(\text{map}()\) as a primitive instruction for LINDA.

Contrary to initial expectations, constraint matching versions of the basic LINDA retrieval instructions have been identified as generic operations. These are generalisations of the existing retrieval instructions, extending the functionality of each retrieval instruction by enhancing the matching process to any given constraint function.

The LINDA implementations of the constraint matching operations employed the \(\text{eval}()\) instruction, which has hitherto tended to be ignored amongst the LINDA instructions.

The \(\text{eval}()\) instruction, when using multiple tuple spaces, allows processing to be migrated to a tuple space where most of the retrievals will be read. This is most useful in a distributed implementation of LINDA where different tuple spaces are stored on different processors. So, if a process interacts a lot with a particular tuple space it is more efficient for that process to be running on the same processor as that which that tuple space is stored on. This is where the process spawning capability of \(\text{eval}()\) presents itself as an ideal means for such process migration, though there is still the matter of distributing that processing if the data are distributed.

Given that one is migrating processes to execute locally in given tuple spaces, this suggests the notion of a default local tuple space for tuple space interaction instructions. Therefore, rather than specifying the name of the tuple space to be interacted with each time, the default case would be for the interaction to take place with the tuple space in which the process is executing as part of an active tuple. This would suggest a greater degree of abstraction available in the writing of processes as a process could be written to interact with its local tuple space, then replicated and executed on many different tuple spaces, each of which is local to that process instance.

Also, the constraint matching generalisation of the basic LINDA retrieval instructions suggests the possibility of replacing the basic retrieval instructions with their more general variants. After all, the basic retrieval instruction is but a special case of the constraint matching variant. This would result in a LINDA with more powerful retrieval instructions.

Furthermore, in the course of implementing the case base retrieval application in LINDA, the bulk retrieval instructions devised at York were found to be essential. They provided the means for replicating tuple spaces so that the numbers of tuples could be counted and tuples consumed uniquely. That is, they provided snap-shots of tuple spaces...
upon which processing could be performed without interfering with the original tuple space.

Further work is being undertaken in incorporating the new operations identified herein into a new LINDA kernel implemented over PVM.

Acknowledgements

This work was undertaken under grant number GR/J99353 in the AIKMS programme of the EPSRC of the UK.

References


