ISETL-LINDA: Parallel Programming with Bags

Andrew Douglas, Antony Rowstron and Alan Wood

Department of Computer Science,
THE UNIVERSITY of York

Abstract. This paper describes the parallel language ISETL-LINDA. The language is an extension of ISETL, an imperative language whose main data structure is the set. We extend ISETL by adding a new type of distributed bags, and operations to act on bags which correspond to the LINDA primitives. However, we use a variant of LINDA created at the University of York which replaces the problematic `inp` and `rdp` primitives by a single primitive, `collect`.

After describing the language, we give an example of its use with a simple case study. We then describe an implementation across a network of processors, outlining the implementation of distributed bags, and our process spawning mechanism. The language has been implemented on a Meiko Computing Surface.

1 Introduction

This paper describes a new language, ISETL-LINDA, and its implementation over a network of transputers. The language itself is similar to another parallel language, ProSet [5]. Both are embeddings of Linda [4] into SETL.

Linda is a model for communication and co-ordination of parallel processes. The basic communication medium is tuple space; a tuple space is a shared bag of tuples, with an associative lookup mechanism. Operations are provided to place tuples into a tuple space, and remove tuples from a tuple space. This is the only way that processes are allowed to communicate.

The differences between ISETL-LINDA and ProSet arise in the choice of primitives to implement. The first difference is the implementation of process spawning. We have chosen `eval` [4], which is Linda's mechanism for spawning processes – a process is spawned to execute an expression the result of which is placed into a tuple space. ProSet uses futures – the result of a process is assigned to a variable. Once the process is started, computation continues in the parent. The parent only blocks if it attempts to access the variable to which the process has been assigned and the process has not yet completed.

The second difference is the inclusion of a new primitive, called `collect` [2]. We intend this primitive to replace the problematic primtives `inp` and `rdp`.

We begin this paper by describing ISETL [6], Gary Levin's Interactive implementation of SETL. We then describe the extension, ISETL-LINDA, and give a case study to show how ISETL-LINDA can be used for parallel computation. Finally we outline the implementation of distributed tuple spaces and our process spawning mechanism.

2 The ISETL programming language

ISETL-LINDA is an extension of the programming language ISETL (Interactive SET Language) [6] [1], which in turn is derived from SETL [8]. It is an imperative programming language, in which the main data structure is the set. In this section, we outline some features of ISETL, before describing our extensions.

2.1 Programming is ISETL

ISETL has two kinds of programs – expressions and statements. Expressions, at their simplest, are the objects of the main data types; multi-precision integers, floating point numbers, reals and so on. Expression are also built using the arithmetical operations such as `+` and `-`. Many of these operations are overloaded for different types (or combinations of types). ISETL is very much an weakly typed language. ISETL statements include printing statements, conditionals and iteration.

The two major data types in ISETL are sets and tuples. We will not describe sets here, as they are not relevant to this paper. Tuples, however, are very important. A tuple is a list of heterogeneously typed objects, surrounded by `[` and `]`, and separated by commas. For example, `[1, true, "String"]` is a tuple of three objects (a triple) whose types are integer, boolean and string. Other ways of forming tuples are by ranging (for example, `[1 .. 5]` evaluates to the tuple `[1, 2, 3, 4, 5]`) or by formation: the tuple `[e(x): x in S | P(x)]` is the tuple created from elements of `S` where `S` is a set or a tuple, that have property `P`. The items in the tuple have the value `e(x)`. So `[x : x in [1..5] | even(x)]` is the tuple `[2, 4]`. The order in which items are pulled from `S` is significant. When `S` is a set, a tuple formation may compute to one of many possible tuples.

We can retrieve the elements of a tuple by indexing. Given a tuple `t`, the expression `t(i)` returns the $\mathtt{i}^{th}$ element of the tuple. So `[1,2,3](2)` evaluates to 2. The expression `[1,2,3](8)` is undefined, which is denoted by the special object `OM`.

Sets are built in a similar way, using the brackets { and }.

2.2 Computation and Functions

Computational statements include assignment, sequencing, conditionals and iteration.

Assignment is performed using `:=`. Assignment may be to variables, such as `x` or `var`, or to fields of a tuple variable. So, if `t` is a tuple, then `t(i) := e` assigns the value of `e` to the $\mathtt{i}^{th}$ field of `t`.

All statements are terminated by a semicolon. This acts to sequence statements.

Conditionals are supplied using an `if ...then ...else ...end if` construct, where the `else` clause is optional.

Looping is performed using `while ...do ...end while` and `for v in F do ...end for`, where `F` is either a tuple or a set. When using a set, the order in which the values in `F` are assigned to `v` is non-deterministic.

ISETL also has functions. For example

```
f := func (a,b);
        ans := a + b;
        return ans;
     end func;
```

assigns to `f` a function of two values, which returns their addition. The statement

```
f := |x,y -> x + y|;
```

has the same effect. Here, `|x -> e(x)|` is similar to $\lambda x \cdot e$ in the lambda calculus. Functions are *first class* and may be passed to, and returned from functions.

3 The ISETL-LINDA programming language

Linda provides primitives for communication between processes. This communication takes place using *tuple spaces*, which are shared memories with an associative lookup mechanism. A tuple space is a *bag* of *tuples*.

ISETL already provides a type of tuples. Our extension will provide two new types: *templates*, which are used in the matching process, are similar to tuples; *tuple spaces* represent distributed bags of tuples, in the Linda sense. We provide functions for the manipulation of tuple spaces in ISETL-LINDA.

3.1 Templates

Corresponding to tuples, we introduce a type of *templates*, which are used in the process of *associative matching*. A template, like a tuple, is an ordered sequence of heterogeneously typed objects. For example, `|[1, true, "fred"]|` is a template of three fields. This template will precisely match the tuple `[1, true, "fred"]`. Templates are different from tuples because a field of a template may be typed, but have no value. For example, we can have a template `|[1, ?bool, "fred"]|`, which matches the tuple above, but will also match the tuple `[1, false, "fred"]`. The ?*type* notation represents a typed hole. Templates are first class objects and may be passed between processes[1].

A template will *match* a tuple if the following hold:

– The tuple and the template have an equal number of fields,

– each field contains an item of the same type,

– each field contains an object with an equal value, or the template field contains a hole.

[1] This feature is experimental, because it is not entirely clear how we can `lin` a template.

3.2 Tuple spaces

Tuple spaces are denoted using `|{` and `}|`. Bags are unordered, and may contain more than one of a particular item. They are also untyped, although may contain only tuples.

Tuple spaces are actually named objects and ISETL-LINDA manipulates tuple space *names* or *references*. Consider the following

```
a := |{}|;
b := a;
```

The first assignment binds a new, empty tuple space to the variable `a`. After the second assignment, the variables `a` and `b` become *references* to the same tuple space – any change made to either variable affects a change of the other.

We describe some functions for manipulating tuple space in the next section. However, for completeness, we allow the creation of new *non-empty* tuple spaces in the manner that one defines sets or tuples. For example, `|{[1], [2], [4]}|` returns a reference to a tuple space which will contain the three tuples listed. We can also use formation to create tuple spaces; for example `|{[x,y] : x in S, y in R | P(x,y)}|` creates a new tuple space containing tuples formed from the carriers `S` and `R` [2].

3.3 Manipulation of tuple space

ISETL-LINDA provides the following functions to manipulate a particular tuple space:

- `lout(ts, t)` takes a tuple space `ts` and a tuple `t`, and places `t` into the tuple space `ts`
- `lin(ts, t)` takes a *template* `t`, and searches in the tuple space `ts` for a *tuple* matching `t`. If a suitable tuple is found, it is removed from tuple space and returned as the result. Otherwise, the process making the call is blocked until a suitable tuple enters the tuple space.
- `lrd(ts, t)` takes a *template* `t`, and searches in the tuple space `ts` for a *tuple* matching `t`. If a suitable tuple is found, a copy is returned as the result. Otherwise, the process making the call is blocked until a suitable tuple enters the tuple space.

  The tuple returned remains in the tuple space.
- `lcollect(ts, tt, t)` takes two tuple spaces, `ts` and `tt`, together with a template `t` and moves tuples in `ts` which match `t`, into the tuple space `tt`. It returns a count of the number of tuples moved.

The `collect` primitive replaces the original Linda `inp` and `rdp` primitives [4] (although they are included for completeness).

[2] At present, tuple spaces cannot be carriers in a formation expression.

3.4 Spawning processes using `leval`

The `leval` primitive is ISETL-LINDA's way of spawning processes. Given a tuple space and an expression, a thread is spawned which evaluates the expression and places the result in the tuple space. The result of the expression, therefore, must be a tuple. For instance, `leval(ts,[e, s, t])` will spawn a new thread to evaluate the tuple `[e, s, t]`. The resulting tuple will be placed in the tuple space `ts`.

In fact, in this case, `leval` must also evaluate the three expressions `e`, `s` and `t` in parallel, if any of the components access tuple space.

3.5 A discussion about process communication and co-ordination

We have stated already that tuple space provides communication between processes. This communication is provided in a highly asynchronous manner. However, we also want tuple space manipulation to provide *co-ordination* between processes. For instance, a process may be waiting upon another to complete some operations on tuple space before it may continue. A common Linda way of performing such co-ordination would be for one process to look for a tuple to indicate that the other has finished. Once this tuple is detected, it is assumed that the results produced by the other process are already in tuple space.

In order to make this assumption, we require that when any process does a sequence of `lout` operations, those operations are done in order, and that the result of an `lout` has entered tuple space before those which follow. For example, take the two code fragments of figures 1 and 2.

```
$ A Fragment of a process which lout's
$ two results and a "Done" tuple.

lout (ts, [tuple1]);
lout (ts, [tuple2]);
lout (ts, ["Done"]);
```

Fig. 1. Fragment of Process One

If the order of the `louts` in process one were to be changed (by, say, an implementation which did not preserve this ordering), it would be possible that when process two detects the presence of the `"Done"` tuple, it could perform the `collect` but not collect all of the remaining tuples. Linda would then be failing to act as a co-ordination language. We could overcome this, at the expense of placing a lock on the tuple space which contains the number of tuples that should be collected. However, such locks are expensive in terms of tuple space accesses,

```
$ Wait for the "Done" tuple to appear

while (collect (ts, ts, |["Done"]|) = 0) do
   perform_some_operations();
end while;

$ Collect all the results and
$perform some work on them

n := collect (ts, tt, |[?int]|);

work_on_tuples (n, tt);
```

Fig. 2. Fragment of Process Two

and so it would be better to enforce within an implementation that the ordering of `out`s is preserved.

Another important aspect has to do with `leval`. Within the Linda model, it is assumed that we have infinite resources, and that a spawned process will begin execution immediately. This, of course, is not the case. However, we must give the illusion that this is so: once a thread is requested using `leval`, it must be in a position to be executed. This has not always been the case in past implementations [9].

An example where this necessity is cleary seen is given diagrammatically in Figure 3. Here, we have a number of threads started using `leval`, which communicate via tuple space in such a way that they form a pipeline. That is, each thread is receiving results from only one thread and passing results to another, through a number of tuple spaces. When all threads are executed concurrently, the pipeline works. If a single thread is not in a position to be executed (say, through lack of resources), then the whole pipeline comes to a halt.

In the next section, we outline an implementation of tuple spaces and threading which behaves correctly with respect to two issues outlined.

4 A case study: Parallel Fold

The following program performs the parallel *fold* of a bag of integers. Given a bag of tuples containing integers, |{[a_1], ... , [a_n]}| and some binary operation, $\oplus$, which is commutative and associative, the *fold* of the operation over the integers will return the tuple

[$a_1 \oplus a_2 \oplus \ldots \oplus a_n$]

The function `FoldTuples` has parameters for the number of integers contained

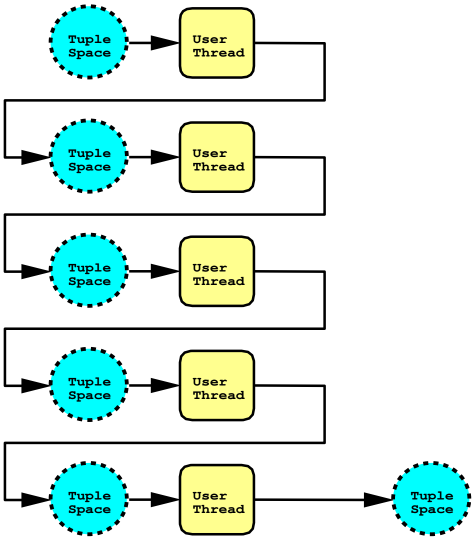
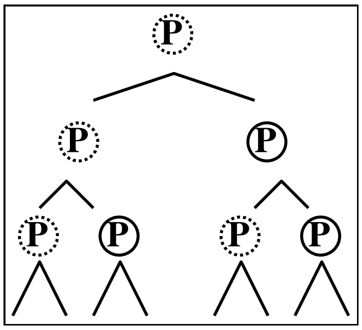


Fig. 3. A pipeline of User Threads, communicating through Tuple Spaces

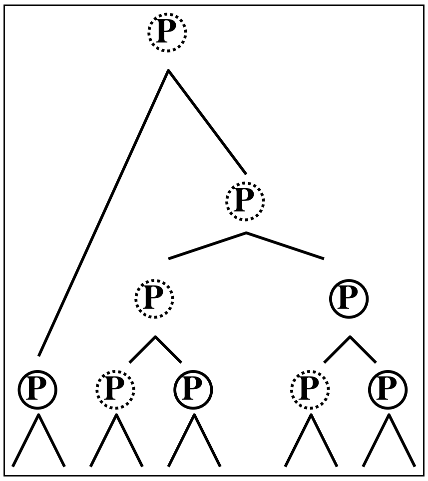
in the bag, the tuple space containing these integers and the operation to be performed. It spawns `(nummy + 1) div 2` threads, to evaluate the function `worker`. Each worker is passed two arguments, which are used to calculate how much work it must perform before it dies voluntarily. The function then waits for all the workers to complete before removing the result from tuple space.

```
FoldTuples := func(nummy,ts, myf);

  local n, ans;

  ans := |{}|;

  for n in [0 .. (((nummy+1) div 2) - 1)] do
    leval(ans,[worker(myf,n,nummy,ts)]);
  end for;

  for n in [1 .. ((nummy+1) div 2)] do
    lin(ans,|["WORKER_TERMINATED"]|);
  end for;

  n := lin(ts,|[?int]|);

  return n;

end func;
```

The function `worker` (see below) repeatedly loops, pulling two numbers from the tuple space, performing the operation on them, and placing the result back in the tuple space. However, once the operation has been performed on each pair of numbers in the tuple space, we are left with half as many numbers to fold, and too many processes. If these processes were left to compute, they might each grab one number from the tuple space, and block waiting for another which is not present.



(a) Folding eight numbers



(b) Folding ten number

Fig. 4. Trees representing folding

We require, then, that approximately half of the remaining processes die. This can be seen in the tree of Figure 4(a), where each circled P represents a process which consumes two numbers and outputs its result. Those Ps with dashed circles die at that level. Here, the fold is performed on eight numbers. We begin with four processes. After the first wave of number have been eaten, two processes die, and two continue to eat the other four numbers. After this, one process dies, and the remaining one performs the operation on the remaining two numbers before dying itself.

However, the decision about whether a process dies or not is not a simple one. Take the tree of Figure 4(b). Here, the fold is operating on ten numbers. We begin with five processes. After the first wave of numbers have been consumed, two processes die, leaving three to consume five numbers. However, worker 0 misses a turn, so that only four numbers are consumed, leaving three in the tuple space. Again, one process dies, one consumes two numbers and dies, and process 0 misses a turn. Finally, process 0 will consume the remaining two numbers before dying itself.

The calculation following the operation establishes whether the thread lives or dies, and uses the second parameter passed to the worker. The conditional around the two `lin` operations decides whether worker 0 is required to miss a turn. This *operate and die* loop continues until only one number remains in the tuple space. The fold is completed once a `["WORKER_TERMINATED"]` tuple has been detected from each of the worker threads.

```
worker := func(fn,position,tuples,ts);

  local work,v1,v2;

  work := 0;

  while work = 0 do
    if ((position /= 0) or (even(tuples))) then
      v1 := lin(ts,|[?int]|);
      v2 := lin(ts,|[?int]|);
      lout(ts,[fn(v1(1),v2(1))]);
    end if;

    tuples := (tuples + 1) div 2;

    if(odd(position) or (tuples = 1)) then
      work := work + 1;
    end if;

    position := position div 2;

  end while;

  return "WORKER_TERMINATED";

end func;
```

5 Implementation

We describe here our implementation of ISETL-LINDA. The description is in two stages. First, we describe the implementation of distributed tuple space. A more detailed explanation is given in [3]. Secondly, we describe in detail our process spawning mechanism. The implementation is for a network of transputers, which poses problems, as we must declare a static process and communication structure. We get around this problem by implementing an ISETL-LINDA engine which spawns threads, and uses local shared memory to communicate internally.

5.1 Implementing Tuple Space

Consider a set of processors, $\{P_1, \ldots, P_n\}$, each with a local memory. We could consider each P_i to be, say, a *T800 Transputer* with 4 Megabytes of memory. We distribute tuple space as a set of *tuple space managers*, $\{T_1, \ldots, T_m\}$, distributed across the P_i is some way. Each T_j consists of a process T_j^P and a queue[3] of requests T_j^Q. The exact processor topology and distribution of tuple space managers is not fixed. However, each tuple space manager must be able to communicate with all the other tuple space managers.

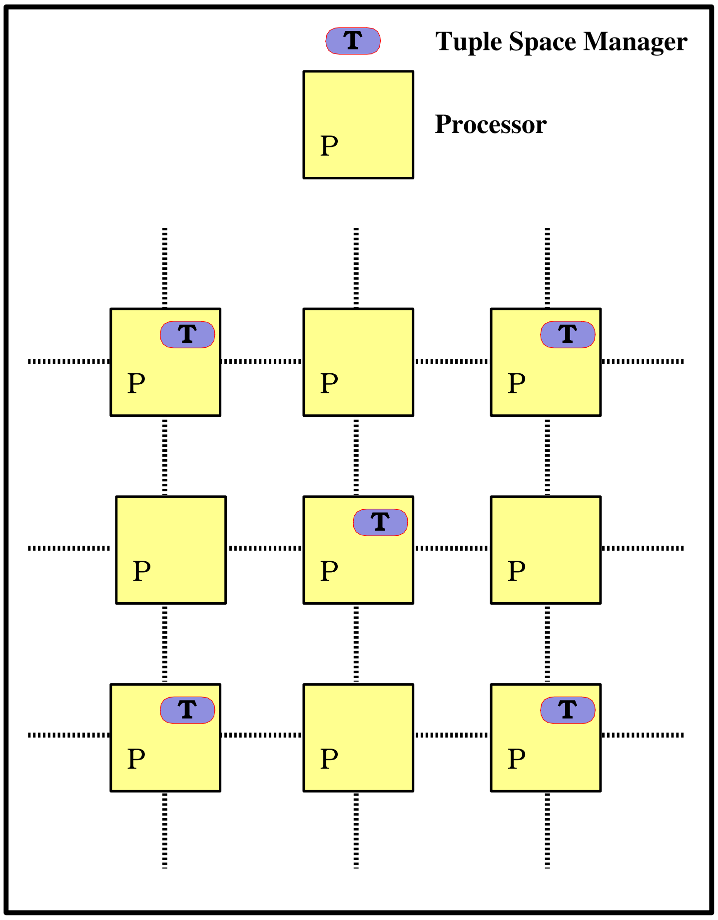


Fig. 5. Distribution of Tuple Space Managers over Processors

User processes make tuple space requests by adding commands to the queue

[3] Which may itself be implemented by a separate process.

of a tuple space manager. We could approach this in two ways. The first requires a user process to know about only one tuple space manager, and for this manager to take care of placement of tuples within the tuple space. The second approach requires a user process to know about all tuple space managers, and for placement to be decided upon by the user process itself. This latter approach has a slight cost advantage, as well as a certain flexibility since placement schemes are compiled with the application, rather than with the tuple space managers.

5.2 Tuple placement

The tuple space is distributed over the set of tuple space managers. Tuple placement and lookup is decided upon by a *pair* of hashing algorithms. The first hashing algorithm uses information which can always be determined, to produce a subset of the tuple space manager processes over which this *kind* of tuple can be placed. For example, given a tuple and a template, we can always determine the number of fields, and their individual types.

The second hashing algorithm uses the result of the first algorithm, plus some specific information which will determine precisely where to place the tuple in question. For a tuple, this information will always be obtainable, but for a template, the information required by the second hashing algorithm may not be determinable. We say that a tuple or template for which this information is determinable is *completely presented*.

As an example, when we perform an `out` operation, the tuple is *completely presented*; it contains no holes. The first hashing algorithm could give us a range of tuple space mangers T_x to T_y over which to place this kind of tuple. Then, the second algorithm could take the value of the first field of the tuple, produce an integer value representation of it, and then return this value, modulo $(y - x)$. Assuming that the value of the first field changes regularly, the distribution of this kind of tuple over the range of tuple space managers will be fairly even.

Performing an `in` operation follows a similar routine. The first algorithm will take a template and produce the range of tuple space managers over which this kind of tuple is distributed. Then, if the template is *completely presented* in its first argument (that is, the first argument is *not* a hole), we can determine precisely which tuple space manager will contain such a tuple. Otherwise, if the template is not *completely presented* in its first argument (that is, the first argument *is* a hole), we can leave requests for this tuple on the queue of all tuple space managers within the range specified by the first algorithm. Some arbitration is required so that only one matching tuple is removed from the tuple space and returned to the user process.

5.3 Blocking processes

Blocked processes (processes waiting for the result of an `in` or `rd` primitive) are kept in a list in the appropriate tuple space manager where a matching tuple is expected to be found. In the case where a template is not completely presented, the process appears in the blocked process list of all the possible tuple space managers where a match may appear. Again, some arbitration is required to ensure that only one matching tuple is returned.

5.4 Some further points

We note a number of things. Firstly, that hashing takes computation time, and we must choose hashing algorithms that take $\mathcal{O}(1)$ time to compute.

Secondly, since we are distributing tuple space across processors, and across tuple space managers, we want to provide hashing algorithms that give a fair and even distribution of tuples, and a distribution which can provide truly concurrent access to tuple space.

Also, hashing is performed on the contents of a tuple, regardless of the name of the tuple space. This is important when we think about implementing the new `collect` primitive. When a `collect` occurs, a collect message is sent to all those tuple space managers in which a tuple match may occur. These tuple space managers then simply re-tag all the matching tuples so that they belong to the destination tuple space. Any processes blocked on these tuples will appear in the blocked process list of the same tuple space manager, and can be awakened as appropriate tuples appear. So `collect` can be implemented in a simple and very efficient way.

Finally, in some compiled languages, where the entire code is presented before runtime static analysis of the code can be performed, to determine the pattern of process communication. This can often result in a very fast Linda runtime system, since appropriate hashing algorithms can be chosen at compile time.

However, in any environment where processes and code are dynamic, such as in interpreted language environments, this kind of analysis cannot take place. In addition to this, we are proposing that tuple spaces and templates are first class objects, which can be passed between processes via tuple spaces. This makes static analysis much more difficult to perform[4]. We can then provide hashing algorithms which in the general case will perform sufficiently well.

5.5 Eval

The `leval` primitive is ISETL-LINDA's way of spawning processes. However, unlike some, who view `leval` as acting *within* a tuple space[5], we take the view that `leval` is simply for spawning processes, and is more to do with the language of computation than it is to do with Linda.

We require, then, an *engine* which, given a computation, can spawn a *thread* which will perform the computation and place the result in a tuple space. More importantly, we want to give the impression of unlimited, dynamic creation of processes. Such an ISETL-LINDA engine consists of the following items:

[4] There is an analogy here with the static analysis of functional languages with higher order functions.

[5] Some propose *active* tuples – tuples which are within tuple space, evaluating themselves into *passive* tuples. Some go as far as to be able to `in` an active process, thus suspending its computation.

- A main function which starts threads, and cleans up after them,
- A communication channel for this function,
- A function which, given a closure[6], will unpack it, execute it and send the result to tuple space,
- A single communication which is shared between threads, together with a process which receives messages and distributes them correctly,
- A global list of runtime elements.

It is a consequence of the transputer architecture that communication links must be declared in advance, before computation begins. Because all threads need a communication channel to the assigned tuple space manager, we must share this channel. The communication channel becomes a *transport*, using Meiko CSN terminology [7], together with a buffer process which stores incoming messages. A semaphore will be required to control access to the transport. When a message is received, it must be tagged with the unique identifier of the thread to which it is sent. The message is placed in some global list of messages. A thread requiring a message waits on a semaphore assigned to its identifier for an incoming message.

The main function waits for messages containing closures. A thread is created using the `runp` function provided by the Meiko CS-Tools environment.

Each spawned thread will have an index to the list of stack pointers, which holds information about the state of its computation. All references to the heap will be embedded within this list, so that, at any time, the thread process can perform garbage collection. It is hoped, in future, that there will be only one heap. However, at present, each thread has its own local heap which is created when the thread is started.

The main process also has the duty of clearing up, once a thread has completed. A marker in the runtime element of a particular thread indicates whether the thread is still running. Any dead thread can be removed, have its heap freed and the runtime element removed.

5.6 Placement of threads

When an `leval` is performed, a closure is created and sent to an engine for execution. In the prototype implementation, the engine to be used is chosen in a round-robin way. Research is in progress to design some other mechanism which would provide a more balanced approach to process placement.

[6] A closure is an expression, together with any code that is required and the environment containing the values, at the time the closure was created, of any free variables.

6 Conclusion

We have described a new parallel language, called ISETL-LINDA, which as embedding of the Linda model into the language ISETL. We gave a case study to show how the language can be used, before describing its implementation over a network of transputers.

6.1 Further Research

It would have been nice to add bags to ISETL, and to implement operations over bags which correspond to the mathematical operations on bags (which have parallels with those of set theory). One can then draw comparisons between the operations on bags and the Linda primitives on tuple spaces.

As examples of how bag operations parallel with Linda operations, take bag membership. This is similar to the Linda `rdp` operation [4]. Bag union can be implemented by performing one or more `lout` operations. However, the difficulty comes when deciding whether the expression

```
b := ts + |{[1], [2], [3]}|        $ + represents bag union
```

is updating the tuple space `ts`, or whether `ts` is copied first and the new items (does this bag exist in its own right?) placed into the new bag. A copying semantics would be consistent with the notation for sets, but reference semantics seems in some way more in tune with the Linda view.

Further research is needed to see if we can further integrate the tuple space operations into ISETL in this way.

6.2 Further Work

We shall be comparing the expressibility of ISETL-LINDA with other high level languages in which Linda has been embedded. Our next implementation will use Gofer, the functional programming language, as the basis of Gofer-LINDA.

Acknowledgements

This work is partially funded by EPSRC Research Grant GR/J12765. Antony Rowstron is supported by an EPSRC CASE Grant with British Aerospace Military Aircraft Division.

References

1. Nancy Baxter, Ed Dubinsky, and Gary Levin. *Learning discrete mathematics with ISETL*. Springer Verlag, 1989.
2. Paul Butcher, Alan Wood, and Martin Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505 – 516, 1994.

3. Andrew Douglas, Antony Rowstron, and Alan Wood. Linda implementation revisited. In 18^{th} *World occam and Transputer User Group Conference*, Transputer and occam Engineering Series. IOS Press, 1995.
4. David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages*, 7(1), January 1985.
5. W. Hasselbring. Prototyping parallel algorithms in ProSet-Linda. In J. Volkert, editor, *Parallel Computation (Proc. Second International ACPC Conference)*, volume 734 of *Lecture Notes in Computer Science*, pages 135–150. Springer Verlag, October 1993.
6. Gary Levin. *An Inroduction to ISETL Version 3.0*, 1990. Distributed with the ISETL source.
7. Meiko Limited, 650 Aztec West, Bristol. BS12 4SD. United Kingdom. *C for CSTools*.
8. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Shonberg. *Programming with Sets: And Introduction to SETL*. Springer Verlag, 1986.
9. Steven Ericsson Zenith. Linda co-ordination language; subsystem kernel architecture (on transputers). Technical Report YALEU/DCS/RR-794, Yale University, 1990.