Optimizing Analysis for First-Class Tuple-Spaces

Suresh Jagannathan*
Department of Computer Science
Yale University
New Haven, CT 06520
jagannathan@cs.yale.edu

Abstract

This paper considers the design and optimization of a simple asynchronous parallel language that uses first-class tuple-spaces as its main communication and process creation device. Our proposed kernel language differs from other tuple-space languages insofar tuple-spaces are treated as true first-class objects. Moreover, we develop a formal framework for constructing an optimizing pre-processor for such a language. The semantic analysis is based on an inference engine that statically computes the set of tuples (and their structural attributes) that can occupy any given tuple-space. The inference system is non-trivial insofar as it operates in the presence of higher-order functions and non-flat data structures (*e.g*, lists). The result of the inference procedure can be used to customize the representation of tuple-space objects.

1 Introduction

Communication and synchronization are fundamental concerns in the design of any parallel language: through what medium is information transmitted from one process to another and how is access to this medium regulated?

In this paper, we consider the semantics and implementation of a simple asynchronous (*i.e.,* loosely synchronized) parallel language that uses *distributed data structures* as its main communication device. A distributed data structure is a shared data object to which many processes may append and remove information. It is a *non-strict* object; thus, it is meaningful for processes to probe its contents even if not all of its component elements are available. Processes communicating via distributed data structures do so with minimal coordination and book-keeping: processes may deposit data without being aware of the receivers who will access it; processes may access data without being aware of the producers that generated it. The semantics of a program in which such requests occur is well-defined. Non-strictness implies asynchrony since the generation of information is decoupled from its consumption.

Distributed data structures are found in a number of languages. Non-strict languages based on *futures* (*e.g.,* [11]) support distributed data structures since a complex structure whose elements are futures may be examined and manipulated by many processes simultaneously. The write-once I-structure [1] found in the mostly-functional language Id [18] can also be viewed as a distributed data structure: an expression that attempts to access an empty I-structure blocks; well-defined constraints are imposed on how such structures are filled. Concurrent object-oriented languages (*e.g.,* [12]) that permit an object to

*This work was supported in part by National Science Foundation grant CCR-8657615 and in part by Office of Naval Research grant N00014-89-J-1906. Appears in Advances in Languages and Compilers for Parallel Processing, MIT Press, 1991.

receive messages simultaneously from several processes also support a form of distributed data structure since the state of such an object may be visible to many processes concurrently.

The most well-developed use of distributed data structures is in the Linda programming model [4]. The fundamental mechanism by which distributed data structures are built and manipulated in Linda is the *tuple-space*, a data abstraction that resembles a shared associative memory. A tuple-space permits processes to add, read, evaluate and remove shared data. (The main data objects which reside within tuple-space are known as *tuples*.) C.Linda, a dialect of C that supports the tuple-space abstraction, has been used with notable success on a variety of different machines and applications [2, 5]. We discuss the semantics of tuple-spaces in the sections following.

A tuple-space is a powerful abstraction because it imposes no pre-defined structural constraints on the elements which occupy it. The tuples which occupy a tuple-space can be assembled into arbitrary kinds of data structures. The elements of these data structure collections can be read by many processes simultaneously. Synchronization is mediated by a unification procedure that attempts to unify a tuple containing unbound variables with an element in a bag of tuples found within a specified tuple-space. Failure to produce a substitution causes the executing process to block.

Although it's a flexible device, its amorphous structure makes its efficient implementation a challenging exercise. Optimizing compilers for C.Linda do exist [6] and the technologies which they employ have been quite successful. Nonetheless, no semantic framework for describing or analyzing these optimizations has yet been developed; our goal in this document is to develop such a framework and to significantly expand on what has been thus far developed only informally. We expect compilers based on these semantic analyses will be simpler to construct and will implement a wider range of optimizations.

The paper is organized as follows. In the next section, we present a kernel language (called $\mathcal{TS}$) whose semantics permit the creation and manipulation of *first-class tuple-spaces*.

Section 3 develops a semantic analysis framework for this language. The analysis is described in terms of an inference engine that statically computes the set of tuples that can occupy any given tuple-space. The inference system is non-trivial insofar as it operates in the presence of higher-order functions and non-flat data structures. The result of the inference procedure is then used to customize the representation of tuple-space objects. Section 4 presents conclusions and scope for further research.

2 A Kernel Language

We first give an informal description of $\mathcal{TS}$ and then discuss the motivation underlying its design. A formal semantics for the language is given in [14].

$\mathcal{TS}$ is defined by the grammar shown in Figure 1. We use x to range over identifiers, s to range over strings, and c to range over natural numbers. `E` (and its subscripted variants) range over expressions; `T` (and its subscripted variants) range over the class of expressions allowable inside tuple-space operations.

2.1 Informal Semantics

The basic operators in $\mathcal{TS}$ form a simple higher-order lexically-scoped functional language with sequencing (;), conditionals ($\rightarrow$) and lists (:). In addition to function definition and application, $\mathcal{TS}$ also supports five operations for creating and manipulating tuple-spaces.

2.1.1 Creating Tuple-Spaces

The `make-ts` operator creates a new tuple-space; more precisely, `make-ts` returns a reference to a tuple-space object. A tuple-space is a data abstraction that defines a shared associative memory whose

E ::= `true` | `false` | c | x | s |
(λ (x_1 x_2 $\ldots x_n$) E) | (E_1 E_2 $\ldots E_n$) |

[ ] | [E:E] | (`hd` E) | (`tl` E) |
$E_b \rightarrow E_t$; E_f |
(`begin` E_1 ; E_2 ; ...; E_n) |

(`make-ts`)
(`out` E (E_1, E_2, ..., E_n))
(`eval` E (E_1, E_2, ..., E_n))
(`rd` E (T_1, T_2, ..., T_n) E_b)
(`in` E (T_1, T_2, ..., T_n) E_b)

T ::= E | $?x$

Figure 1: Grammar for $\mathcal{TS}$.

elements are heterogeneous ordered sets of values known as *tuples*. The value yielded by `make-ts` is a *tuple-space*. A tuple-space in $\mathcal{TS}$ is a first-class object that may be built into list structures, passed as an argument to a function or returned as a result of an application.

The value yielded by `make-ts` can be thought of as a *capability* to a shared associative memory; expressions that have access to this capability can perform read, write and remove operations on this memory. This model suggests a non-hierarchical relation among tuple-space: the elements of a tuple-space are tuples whose component fields are themselves never tuple-space objects. A tuple can contain the address of a tuple-space, never the tuple-space itself.

The lifetime of a tuple-space is a function of the number of objects and variables which refer to it. A tuple-space that is not referenced by any active object[1] or variable may be garbage collected. Thus, in the expression:

```
( (λ (x) (λ (y) y)) (make-ts) )
```

the lifetime of the tuple-space bound to `x` is precisely the lifetime of `x`; once the outer application is evaluated, the space allocated for this tuple-space may be reclaimed.

A process can communicate with other processes only via the set of tuple-spaces to which it has access. (We describe how processes are created below.) In other words, processes in this computation model cannot communicate via environments or ordinary shared variables; the only shared data object in this model are tuple-spaces. Figure 2 illustrates the relationship between tuple-spaces and binding environments.

2.1.2 Access and Manipulation of Tuple-Spaces

A tuple is deposited into a tuple-space using one of two operators. Let `ts` be a tuple-space[2]; then, the expression:

(`out ts` (e_1, e_2, ..., e_n))

evaluates each of the e_i to get a value v_i and deposits the tuple,

[1] An object is "active" if it is be referenced by some other active object.

[2] In the following, we'll blur the distinction between a reference to a tuple-space and the tuple-space itself when the intended meaning is clear from context.

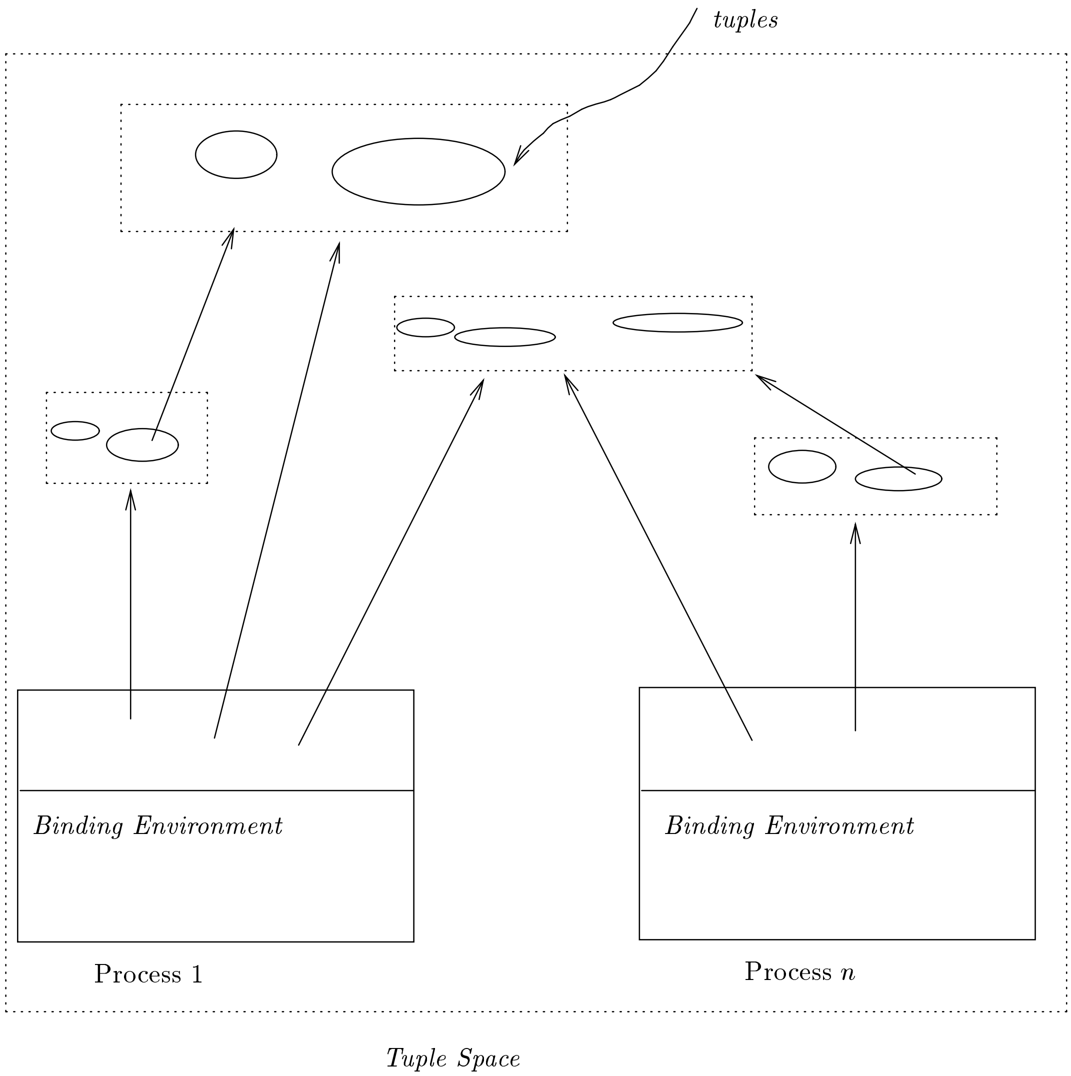

Figure 2: The relationship between tuple-space and binding-environments.

[v_1, v_2, ..., v_n]

into the tuple-space denoted by `ts`. A value is either a constant (*e.g.*, an integer, Boolean or string), a list whose elements are all values, a closure[3], or a reference to a tuple-space object. `Out` is strict in all of its arguments; thus, each of its arguments must evaluate to values before the operation is considered complete. The value of `out` is unspecified; it is executed for effect, not value.

The e_i in the tuple to be generated are ordinary $\mathcal{TS}$ expressions. They may include identifiers, applications, function definitions, or other tuple operations. For example, the expression:

```
((λ (x) (out ts (x, 10)) (make-ts))
```

deposits a two-field tuple into the tuple-space denoted by `ts`. The first field in this tuple is a reference to a tuple-space; the second field is 10.

`Eval` is the non-strict counterpart of `out`. If `ts` is a tuple-space, the expression

(eval ts (e_1 e_2 ... e_n))

deposits a tuple consisting of n processes into `ts`; the i^{th} process is responsible for computing the value of e_i. Each process evaluates within its own private environment. To achieve this, all functions applied within `eval` are transformed into closed combinators that reference no free variables.

`Out` and `eval` generate tuples into tuple-space. `Rd` and `In` read tuples and binding-values from tuple-space. Consider the expression:

(rd ts (e_1, e_2, ..., e_n) *body*)

Unlike tuple-generator expressions, each of the e_i may be either an ordinary $\mathcal{TS}$ expression *or* an identifier prefixed by a "?". The evaluation of the above expression takes place in two steps. Every $\mathcal{TS}$ expression is first evaluated to a value. The resulting object is known as a *tuple-template*: it consists of values and identifiers prefixed with "?"; these identifiers are known as *formals*.

This template is pattern-matched against the tuples found in `ts`. A successful pattern-match results in each of the "?"-prefixed identifiers in the template obtaining as their binding-value the value of the corresponding field in the tuple to which this template was matched. These bindings are then used in the evaluation of *body*. The result of the entire expression is the value yielded by *body*. Free references in *body* not defined by any of the formals are resolved relative to the `in` or `rd`'s lexical environment.

Assume that the tuples,

[true, 10, *closure of* (λ (x) (1+ x))]

and

[false, 20, *closure of* (λ (x) (1+ x))]

have been deposited into `ts`. The expression:

```
(rd ts (true, ?x, ?y) (y x))
```

yields 11. The tuple-template successfully pattern-matches against the first tuple; as a result of this match, the variable `x` becomes bound to `10`, and the variable `y` becomes bound to the closure. This template does not match the second tuple because of the different values found in the first field of the template and the tuple.

If the pattern-matching operation fails because there is no tuple with the same value-correspondence of actuals, the operation blocks. It resumes when a tuple of the appropriate structure and/or the requisite binding-values have been deposited. If there are several tuples that match a template, only one is chosen to establish the binding-values of the template's formals.

[3] A closure is pair consisting of the values of all free variables defined within the function along with the text of the function body.

The `in` operation is semantically identical to `rd` except that the tuple chosen for the match is removed from the given tuple-space. Thus, the expression

```
(in ts (true, ?x, ?y) (y x))
```

establishes the appropriate bindings and removes the tuple

[true, 10, *closure of* λ (x) (1+ x))]

before evaluating the application constituting its body.

2.2 Why First-Class Tuple-Spaces?

First-class tuple-spaces have been proposed elsewhere [9, 13] and have been implemented (to a degree) in certain implementations of Linda [16]. The kernel language developed here extends these proposals by embedding first-class tuple-space into a language that supports higher-order functions and complex structure types. The matching algorithm over tuples is suitably extended to handle tuple-spaces as well.

The motivation for incorporating first-class tuple-spaces is in part technical and in part conceptual. By permitting tuple-spaces to be denoted, we allow the programmer to partition the communication medium as he sees fit. Partitioning of this kind in turn permits the compiler to customize the representation of individual tuple-spaces based on their contents. Of course, a clever optimizing compiler can infer a partitioning scheme even if tuple-spaces were not explicitly denotable. Such a compiler would require the use of a designated field within a tuple to serve the role of a hash key — all tuples containing the same hash key would be placed in the same partition of tuple-space. First-class tuple-spaces simplify the complexity of compile-time analysis in this respect. Moreover, the specification of the optimizations underlying the parameterization of a tuple-space structure becomes amenable to formal description because tuple-spaces themselves constitute a distinguished type: the same analyses underlying the optimization and use of other data types in the base language can be applied to tuple-spaces as well.

Conceptually, first-class tuple-spaces unify the conceptual bridge between the ordinary data objects found in the base language and the processes and passive objects resident within a tuple-space. Without first-class tuple-spaces, it becomes difficult (if not impossible) to denote a well-defined subset of the objects and processes resident in tuple-space using the linguistic mechanisms available within the base language. Modularity is therefore compromised: all tuples are accessible to all expressions; conventional namespace management techniques applicable over objects in the base language cannot be applied to the objects found in tuple-space. Ad hoc conventions are invariably required to impose structure over the global tuple-space. Greater modularity and control over process structure are the primary motivating factors in the design of the kernel language described here; both concerns are addressed by first-class tuple-spaces. To build a set of related processes, we encapsulate the processes within the same tuple-space; data values that need to be accessed by a known collection of processes are deposited within a tuple-space accessible only to these processes.

The absence of any rigid structural constraints on the contents of a tuple-space has other implications as well. Since tuple-spaces contain processes (generated via `eval`) as well as passive data elements (generated via `out`), there is no pre-specified process structure imposed on a $\mathcal{TS}$ program: $\mathcal{TS}$ programs configure themselves automatically and dynamically based on the number of tuple-spaces created and the contents found within them.

Example 1

Given first-class tuple-spaces it is clear that we can express state change in our kernel language despite the fact that no explicit assignment operation over base language variables is present. We argue that tuple-spaces permit a "higher-level" abstraction for expressing mutable objects in parallel programs than conventional assignment operations[4]. For example, a mutable variable x can be represented as a

[4] "Higher-level" in the sense that a tuple-space encapsulates a collection of mutable objects all of whom may be altered *in toto*. The bag of passive tuples which occupy a tuple-space is the structured analog of a conventional shared mutable memory.

```
I-structure-create =
  (λ ()
      let I-structure = (make-TS)
      in
       (begin
          (out I-structure ("flag",false));
          I-structure))

I-structure-read =
   (λ (I-structure)
       (rd I-structure ("val",?v) v))

I-structure-write =
   (λ (I-structure v)
       (rd I-structure ("flag",?s)
           s = true →
           error;
           (begin
              (out I-structure ("flag",true));
              (out I-structure ("val",v)))))
```

Figure 3: A tuple-space implementation of an I-structure abstraction.

tuple-space; to update x's value, we remove its current contents using `in` and deposit a new value using `out`:

```
(in x (?v)
  (out x new-value))
```

Assignment is basically syntactic sugar for the above sequence of tuple-operations. We give a full description of a cell abstraction in figure 5. Given the optimizing analysis described in section 3, this code fragment would be compiled into basically the same code as would have been generated had we used ordinary shared variables accessed, for example, via semaphores. The advantages of using tuple-spaces here are two-fold: (1) it allows us to retain a well-defined functional core about which we can reason formally; (2) more importantly, different structured-assignment paradigms (*e.g.,* I-structures [1], read-only logic variables [20], accumulators [18], etc.) can be expressed using simple and easily implemented patterns of tuple operations.

To make our discussion more concrete, consider the implementation of an I-structure, a write-once shared data object in our kernel language. We'll model an I-structure as a tuple-space. Whenever a write on an I-structure takes place (*i.e.,* whenever a tuple is deposited), a flag is set within the tuple-space denoted by the structure indicating this fact; subsequent attempts to write into this structure (*i.e.,* depositing further tuples) raises an error; processes that attempt to read an I-structure before a value has been written block as per the normal blocking rules governing tuple access. The code can be expressed in a sugared version of $\mathcal{TS}$ as shown in figure 3.

Example 2

Figure 6 gives a more detailed example. It provides a $\mathcal{TS}$ implementation of pipeline-based version of the Sieve of Erasthosenes prime-number finder algorithm.

The program implements the prime number filtering program as a pipeline in which each pipeline component represents a process dedicated to filtering out all multiples of a particular prime number; each successive component of the pipeline sees only those elements that are not multiples of primes seen earlier in the pipe. Whenever the current last component in the pipeline sees a number that is not a multiple of the prime it represents (*i.e.,* whenever it encounters a new prime), it extends the pipeline by adding a new component process responsible for filtering out multiples of this new prime.

The function `primes` takes as its input two integers, the first indicating the size of the integer sequence to be examined and the second indicating the number of primes desired in that sequence. (We assume that `size` $>$ `n`.) It defines two tuple-spaces: the first is the `primes-list` which holds two-tuples of the form $[i, v]$ where v is true if i is prime and false otherwise. The second tuple-space holds all active pipeline processes.

There are two abstractions used in the program; the first is a stream abstraction shown in figure 4; the second is a cell abstraction shown in figure 5. Streams are implemented as tuple-spaces that processes use to transmit information; new elements can be added to the stream in constant time.

Local to each process is an `end-of-pipe?` flag that is true if the process is the last element currently in the pipeline. The "heart" of each process is a loop function that scans down its input sieve. If the first number in the sieve is a multiple of its local prime, the process loops on the remainder of its input; if the number is not, then there are two possibilities. If the process happens to be the current last component in the pipeline, then the number it has just scanned must be a prime; in this case, it records the fact in the `primes-list`, adds a new pipeline element (*i.e.,* deposits a new process in `active-tuples`) whose filter agent is the prime number just scanned, sets its own `end-of-pipe` flag to nil and, finally, recurses. If the function is not part of the last pipeline element, it attaches the scanned number to its output stream (called `outbox`) and recurses; the loop function found in the pipeline element to the right of this component uses this stream as its input sieve.

Tuple-spaces are used in this example in three different ways: (1) they serve as the main stream communication mechanism between different filters; (2) they serve to implement a cell abstraction[5]; and (3) they act a process control device in which new processes are evaluated. Constructing an efficient inter-process communication protocol given the presence of a large, possibly dynamic, number of processes (as in this example) becomes first and foremost an exercise in modularity; first-class tuple-spaces simplify the task considerably.

3 Semantic Analysis

In this section we consider the problem of generating efficient representations of first-class tuple-spaces based on a static analysis of a $\mathcal{TS}$ program. To see why this is a non-trivial problem, consider the following expression:

```
((λ (x)
      (begin
         (out x ("foo"))
         (out x ("bar"))
         (rd  x (?y) y)))
   (make-ts))
```

The argument to the abstraction in the application is a tuple-space; the body of the function effects a mutation on this tuple-space by depositing two tuples (via `out`). The issue at hand is whether we can statically construct a representation for the argument tuple-space based on properties which can be inferred about the tuples deposited within it. Stated another way, we would like to infer a representation for a tuple-space based on the structure of the tuples which inhabit it.

In the above example, it is clear that the only tuples which occupy the tuple-space denoted by `x` are tuples of length one with the sole tuple-field containing a constant. An ideal representation for a tuple-space of this sort is in terms of two semaphores named `foo` and `bar`: the translation of an `out` expression is given in terms of a V operation on its specified semaphore argument; `rd` and `in` expressions implement P operations. A naive implementation (*i.e.,* an implementation insensitive to the structure of the tuples inhabiting a tuple-space) would consider a most general representation for `x`, namely one in which `x` is

[5] In this example, the need for a cell abstraction would be obviated given an explicit assignment operation.

```
make-stream =
   (λ ()
        let new-stream = (make-ts)
        in
         (begin
            (out new-stream ("head",0));
            (out new-stream ("tail",0));
            new-stream))

stream-car =
  (λ (stream)
       (rd stream ("head",?index)
           (rd stream (index,?value)
               value)))

stream-cdr!  =
  (λ (stream)
       (in stream ("head",?index)
           (begin
              (out stream ("head",(1+ index)));
              stream)))

attach =
  (λ (stream elt)
       (in stream ("tail",index)
           (begin
              (out stream (index,elt));
              (out stream ("tail",(1+ index)));
              stream)))
```

Figure 4: A stream abstraction using first-class tuple-spaces.

```
make-cell =
  (λ ()
       let cell = (make-ts)
       in
        (begin
           (out cell ("state", "empty"));
           cell))

read-cell =
  (λ (cell)
       (rd cell ("value",?v) v))

write-cell =
  (λ (cell v)
      (in cell ("state",?condition)
          condition = "empty" →
            (begin
               (out cell ("state","full"));
               (out cell ("value",v)));
            (in cell ("value",?old-v)
                (out cell ("value",v)))))
```

Figure 5: A cell abstraction using first-class tuple-spaces. We introduce a "state" condition to determine whether an `in` operation should be performed before writing a new value into the cell.

represented by a hash table or some similar data structure suitable for general matching. The constants `foo` and `bar` would serve as keys into this structure. Clearly, this is significantly more complex and inefficient than the semaphore-based representation.

To take another example, consider the expression:

```
((λ (x y) [ x : y ]) (make-ts) (make-ts))
```

Both `x` and `y` are bound to tuple-spaces and are used as elements of a list object. Operations on these tuple-spaces take place indirectly through list access operations. An optimized representation for these tuple-spaces must be sensitive to this fact. A naive implementation in terms of a complex hash structure for these tuple-spaces may be excessively wasteful especially if the tuple-operations performed on these objects require minimal matching.

In order to construct an optimized representation for tuple-spaces, we need to design a procedure capable of computing the *types* (or attributes) of all tuples that can occupy any given tuple-space object. Given such information, it now becomes possible to analyze various structural properties of the set's elements thereby allowing an optimizer to build a suitable representation. The construction of such sets is complicated by the fact that tuple-spaces are first-class objects.

In a system such as C.Linda, an analysis of this sort takes place by considering every tuple to be an element of the single global tuple-space and proceeding to partition this set based on obvious constraints (*e.g.,* tuple-length and type). In $\mathcal{TS}$, on the other hand, not every tuple need reside in every tuple-space. Moreover, the fact that one can abstract over tuple-spaces, return them from functions or embed them inside data structures implies that a given tuple may be deposited into more than tuple-space. Naive examination of the source text will not reveal accurate tuple usage in such cases.

We approach the problem of statically inferring a suitable structure for tuple-space as a variant of a semantic analysis technique known as *collecting interpretation* [15]. Collecting interpretation is an abstract interpretation [7] that attempts to infer the set of values a given variable can acquire during the execution of a program. In our case, the variables of interest are those bound to tuple-spaces, and the values we are interested in are the values of tuple fields.

```
primes =
  (λ (size n)
       let primes-list = (make-ts)
           active-tuples = (make-ts)
           int-stream  = (λ (initial-stream stream first last)
                              let loop = (λ (stream m)
                                            (begin
                                               (> m last) →
                                                  stream;
                                                  (attach stream m);
                                               (loop stream (1+ m))))
                              in
                               (loop initial-stream first))

           pipeline-elt =
             (λ (my-prime sieve)
                  let outbox = (make-stream)
                      end-of-pipe? = (make-cell)
                  in
                   (begin
                     (write-cell end-of-pipe? true);
                     let loop =
                      (λ (sieve)
                           let next-candidate = (stream-car sieve)
                           in
                            (multiple? next-candidate my-prime) →
                               (begin
                                  (out primes-list (next-candidate,false));
                                  (loop (stream-cdr! sieve)));
                               (begin
                                  (read-cell end-of-pipe?) →
                                     (begin
                                        (out primes-list (next-candidate,true));
                                        (eval active-tuples
                                              ((pipeline-elt next-candidate
                                                    outbox)));
                                        (write-cell end-of-pipe? false);
                                        (loop (stream-cdr! sieve)));
                                      (begin
                                         (attach outbox next-candidate)
                                         (loop (stream-cdr! sieve))))))
                      in
                       (loop sieve))

       in
        (begin
           (out primes-list (1,t));
           (out primes-list (2,t));
           let initial-stream = (make-stream)
           in
            (begin
              (eval active-tuples ((pipeline-elt 2 initial-stream)));
              (eval active-tuples ((int-stream initial-stream 3 size))))
           primes-list))
```

Figure 6: A Sieve of Erasthosenes-based prime-number finder using first-class tuple-spaces.

$$
\begin{array}{lll}
\tau & ::= & \sigma \mid \iota \mid \phi \mid \tau_1 \times \tau_2 \times \ \ldots\ \times \tau_n \ \rightarrow\ \tau \mid \mathbf{List}[\tau] \mid TS \\
 & & \\
TS & ::= & TS_\perp \mid TS[Tuple] \\
Tuple & ::= & (\mathcal{K}, \mathcal{C}, \mathcal{F}, \mathcal{S}, \mathbf{N}) \\
 & & \\
\mathcal{K} & ::= & \text{READ} \mid \text{GEN} \\
\mathcal{C} & ::= & \mathcal{C}_\perp \mid \mathcal{C}[\mathbf{N} \mapsto\ E_s \times\ \iota] \\
\mathcal{F} & ::= & \mathcal{F}_\perp \mid \mathcal{F}[\mathbf{N} \mapsto\ \tau] \\
\mathcal{S} & ::= & \mathcal{S}_\perp \mid \mathcal{S}[\mathbf{N} \mapsto\ \tau]
\end{array}
$$

Figure 7: The Type Language

Our approach is as follows. We construct an inference system that associates with every expression a *type*. The type of a tuple-space is defined as a set of *tuple-types* where a tuple-type defines relevant structural properties of a tuple, *i.e.,* length, kind (*e.g.,* read or generate), number and position of formals, actuals and constants. The inference system defines a collecting interpretation in the sense that a well-typed program has a type associated with every tuple-space that defines the structural properties of the tuples which can occupy that tuple-space. [3] gives a general discussion on type inference systems.

Operationally, the inference procedure can be thought of as a constraint system that imposes restrictions on a tuple-space's structure based on the context in which it occurs. Every tuple-space operation contributes to the type of a tuple-space: a tuple-space operation defines constraints on its argument tuple-space based on the constants, formals and actuals its argument tuple contains.

3.1 Preliminaries

Our inference system associates every $\mathcal{TS}$ expression with an expression in a type language defined by the grammar shown in Figure 7. Throughout the rest of the paper, we use the following naming conventions: we assume τ ranges over *types* and σ ranges over *type variables.* $\mathbf{N}$ ranges over integers, $\mathbf{S}$ ranges over strings, and $\mathbf{B}$ ranges over Booleans. ι ranges over all constants and E ranges over $\mathcal{TS}$ expressions. δ ranges over *tuple-types* and γ ranges over *tuple-space* types. We introduce the special ground type ϕ to denote the type of `out` and `eval` expressions. (Recall that these expressions have unspecified value and are executed for effect.)

There are three type constructors in our type language: $\times$ to build cross-products, $\rightarrow$ to build function types and **List** to construct list types.

A type can be either a ground type (*e.g.,* an integer, Boolean or string), a list type, a function type a tuple-type or tuple-space type. List types denote collections of homogeneous values (values of the same type); function types are used to type λ-abstractions.

A tuple-space type is defined as a set of tuple-types as described in figure 8

3.2 Notation

We abbreviate a tuple-space type of the form:

$$TS_\perp[\delta_1][\delta_2]\ldots[\delta_n]$$

as:

- $\mathcal{K}$: the *kind* of tuple being described — READ if the tuple is associated with an `rd` or `in` expression; GEN if the tuple is associated with an `out` or `eval` operation. We write $\mathcal{K}(\delta)$ to denote the kind component of tuple-type δ.

- $\mathcal{C}$: the *constants* found within the tuple being described. $\mathcal{C}_\perp$ denotes tuples with no constants; a tuple-type with $\mathcal{C}$ component:

$$\mathcal{C}_\perp[i_1 \mapsto (E_{i_1} \times \iota_{i_1})][i_2 \mapsto (E_{i_2} \times \iota_{i_2})] \ldots [i_k \mapsto (E_{i_k} \times \iota_{i_k})]$$

  denotes a tuple that has E_{i_j} with associated type τ_{i_j} in position i_j, $1 \leq j \leq k$.

$$\mathcal{C}(i_j) \downarrow 1$$

  denotes E_{i_j} and

$$\mathcal{C}(i_j) \downarrow 2$$

  denotes τ_{i_j}.

- $\mathcal{F}$: the *formals* found within the tuple being described. $\mathcal{F}_\perp$ denotes tuples with no formals; a tuple-type with $\mathcal{F}$ component:

$$\mathcal{F}_\perp[i_1 \mapsto \tau_{i_1}][i_2 \mapsto \tau_{i_2}] \ldots [i_k \mapsto \tau_{i_k}]$$

  denotes a tuple that has formals in positions $i_1, i_2, \ldots, i_k$ with associated type $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_k}$.

- $\mathcal{S}$: the *actuals* found within the tuple being described. $\mathcal{S}$ has the same structure as $\mathcal{F}$.

- the last component of a tuple-type indicates the size of the tuple. If δ is a tuple-type, then $\mathcal{L}(\delta)$ gives the value of its size component.

Figure 8: The component elements of a tuple-type.

$$TS(\delta_1, \delta_2, \ldots, \delta_n)$$

A similar notation applies over components of tuple-types. E.g.,

$$\mathcal{F}(i_1 \mapsto \tau_{i_1}, \ldots, i_k \mapsto \tau_{i_k})$$

defines a tuple whose i_j^{th} field contains a formal with type τ_{i_j}.

Finally, if δ is a tuple-type, then $\Pi(\delta)$ denotes a set of types, $\{\tau_1, \tau_2, \ldots, \tau_n\}$); we write $\Pi(\delta)[i]$ to denote τ_i. We define Π as follows: if

$$\delta = (\mathcal{K}, \mathcal{C}, \mathcal{F}, \mathcal{S}, n)$$

then

$\mathcal{C}(i) \downarrow 2 = \tau_i$, if $\Pi(\delta)[i] = \tau_i$.
$\mathcal{F}(j) = \tau_j$, if $\Pi(\delta)[j] = \tau_j$.
$\mathcal{S}(k) = \tau_k$, if $\Pi(\delta)[k] = \tau_k$.

(i, j, and k range over disjoint sets of natural numbers with the constraint that $i + j + k = n$.)

3.3 Subtyping

Tuple-space types are ordered under a subtype relation. Intuitively, we think of a type τ_1 as being a subtype of τ_2 if τ_2 is more constrained than τ_1. Stated another way, type τ_1 is a subtype of τ_2 if an expression of type τ_1 can be used in any context where an expression of type τ_2 is allowed.

The subtyping rule permits us to constrain the type of a tuple-space based on the expressions which operate over it. For example, if `x` is a tuple-space, the expression:

`(out x` *some tuple*`)`

can be thought of as imposing a constraint on `x` (namely, that `x` must contain *some tuple*). This constraint can be couched in terms of a subtype relation which requires `x`'s type to be a subtype of the type associated with a tuple-space (call it T) that contains *some tuple* as its sole element. Moreover, satisfaction of this constraint requires that `x`'s type contain information regarding the structure of *some tuple*; if `x` can be used wherever T can, then `x` must contain at least as much information as found in the type of T.

If γ_1 and γ_2 are two tuple-space types, then we say that $\gamma_1 \leq \gamma_2$ if $\gamma_2 \subseteq \gamma_1$. In other words, if the set of tuple-types denoted by γ_1 is a superset of the set of tuple-types denoted by γ_2, then γ_1 is a subtype of γ_2 – it can be used in any type context where γ_2 can.

3.4 Type Environments

In presenting our inference rules, we use the symbol A to represent the current type environment. We use the notation

$$A \vdash e : \tau$$

to indicate that expression e has type τ given the type bindings defined in A.

If A is a type environment, then the type expression $A[x \mapsto \tau]$ defines a new type environment:

$$A[x \mapsto \tau](y) = \begin{cases} A(y) & \text{if } y \neq x \\ \tau & \text{otherwise} \end{cases}$$

3.5 Inference Rules

The axioms and inference rules for the type system are presented in a form similar to Gentzen's calculus of sequents[10]. Each inference rule consists of a set of statements called the *antecedents* and a statement called the *consequent*. In writing an inference rule, we separate antecedents and consequents by a horizontal line; axioms have no antecedents and no horizontal line is drawn.

(Type Variable Instantiation)

$$\frac{A \vdash e : \tau}{A \vdash e : \tau[\sigma/\tau_1]}$$

If τ and τ_1 are types and σ is a type variable, then the expression, $\tau[\sigma/\tau_1]$ replaces all free occurrences of σ in τ by τ_1.

(Subtype)

$$\frac{A \vdash e : \gamma}{A \vdash e : \gamma'} \qquad \gamma \leq \gamma'$$

(Identifiers)

$$A \vdash x : A(x)$$

(Abstraction)

$$\frac{A[x_i \mapsto \tau_i] \vdash e_b : \tau_b}{A \vdash (\lambda\ (x_1\ x_2\ \ldots\ x_n)\ e_b) : (\tau_1 \times \tau_2 \times \ldots \times \tau_n) \rightarrow \tau_b} \qquad 1 \leq i \leq n$$

(Application)

$$\frac{\begin{array}{c} A \vdash e_\lambda : (\tau_1 \times \tau_2 \times \ldots \times \tau_n) \rightarrow \tau_f \\ A \vdash e_i : \tau_i \end{array}}{A \vdash (e_\lambda\ e_1\ \ldots\ e_n) : \tau_f} \qquad 1 \leq i \leq n$$

(Conditional)

$$\frac{\begin{array}{c} A \vdash e_b : \mathbf{B} \\ A \vdash e_t : \tau \\ A \vdash e_f : \tau \end{array}}{A \vdash e_b \rightarrow e_t\ ;\ e_f : \tau}$$

(List)

$$\frac{\begin{array}{c} A \vdash e_1 : \tau \\ A \vdash e_2 : \tau \end{array}}{A \vdash [e_1 : e_2] : \mathbf{List}[\tau]}$$

(Tuple-Space)

$$\frac{\sigma \text{ a fresh type variable}}{A \vdash \texttt{make} - \texttt{ts} : \sigma}$$

(Generate)

Three auxiliary definitions are used in the type rules for tuple-space operations:

Definition 1 Let A be a type environment and let $t = (e_1, e_2, \ldots, e_n)$ be tuple expression. Then,

i. $Constants(A, (e_1, e_2, \ldots, e_n)) = \mathcal{C}(i_1 \mapsto (e_{i_1} \times \tau_{i_1}), i_2 \mapsto (e_{i_2} \times \tau_{i_2}), \ldots, i_k \mapsto (e_{i_k} \times \tau_{i_k}))$
if $e_{i_1}, e_{i_2}, \ldots, e_{i_k}$ are constants found in t and $A \vdash e_{i_j} : \tau_{i_j}$ for $1 \leq j \leq k$.

ii. $Formals(A, (e_1, e_2, \ldots, e_n)) = \mathcal{F}(i_1 \mapsto \tau_{i_1}, i_2 \mapsto \tau_{i_2}, \ldots, i_k \mapsto \tau_{i_k})$
if $e_{i_1}, e_{i_2}, \ldots, e_{i_k}$ are the formals found in t and $A \vdash e_{i_j} : \tau_{i_j}$ for $1 \leq j \leq k$.

iii. $Actuals(A, (e_1, e_2, \ldots, e_n)) = \mathcal{S}(i_1 \mapsto \tau_{i_1}, i_2 \mapsto \tau_{i_2}, \ldots, i_k \mapsto \tau_{i_k})$
if $e_{i_1}, e_{i_2}, \ldots, e_{i_k}$ are the actuals found in t and $A \vdash e_{i_j} : \tau_{i_j}$ for $1 \leq j \leq k$.

Given these definitions, our type rule for `out` and `rd` operations are given as follows (the rules for `eval` and `in` are defined similarly):

$$\frac{\begin{array}{c} A \vdash e_T : \gamma \\ t = (e_1, e_2, \ldots, e_n) \\ TS(\text{GEN}, Constants(A,t), Formals(A,t), Actuals(A,t), n) = \gamma \end{array}}{A \vdash (\texttt{out}\ e_T\ (e_1, e_2, \ldots, e_n)) : \phi}$$

(Read)

$$\frac{\begin{array}{c} A \vdash e_T : \gamma \\ t = (e_1, e_2, \ldots, e_n) \\ F = Formals(A,t) = \mathcal{F}(i_1 \mapsto \tau_{i_1}, \ldots, i_k \mapsto \tau_{i_k}) \\ TS(\text{READ}, Constants(A,t), F, Actuals(A,t), n) = \gamma \\ A[e_j \mapsto \tau_j] \vdash e_b : \tau \qquad j = i_1, \ldots, i_k \end{array}}{A \vdash (\texttt{rd}\ e_T\ (e_1, e_2, \ldots, e_n) e_b) : \tau}$$

3.6 Discussion

The inference rules given above do not define a type-checking algorithm; there are, in fact, many possible types that can be deduced for the same expression given different typing algorithms. Provided that the inference system is sound (*i.e.,* it does not yield a type for an expression inconsistent with the type of the expression's denotation), these types are all refinements of some principal (or most general) type. In our type-system the principle type of a tuple-space T is a set of tuple-types that contain precisely the tuples deposited into or read from T; the principal type defines a "minimal" type for an expression[6] We give a type assignment algorithm based on the inference rules given above in Section 3.6.1.

A `make-ts` operation has a type variable as its type. A type variable is a variable that can be instantiated to a more specific type based on the type contexts in which it occurs. In our type system, a type variable can acquire a type that satisfies the constraints imposed by the type contexts in which its associated tuple-space occurs.

Constraints on type variables take place in the type rules for tuple operations. In order for `out` or `eval` operation to be well-typed we require that the type of its argument tuple-space (call this type γ) be the same as the tuple-space containing the tuple found as the operator's second argument. In order for such a constraint to be satisfied, the type of the argument tuple-space must be coercible (via the subtype rule) to γ. In other words, the expression denoting the argument tuple-space must have a tuple-space type that contains at least as many elements as defined by γ. Such a condition would permit application of the subtype rule on that type that would equate it with γ. A similar analysis holds over `in` or `rd` operations.

Consider the tuple operations applied to variable `sieve` in the prime number finder shown earlier. `Sieve` is defined as a tuple-space and is applied at various points to all four stream operators: `make-stream`, `stream-car`, `stream-cdr!` and `attach`. The structural type analysis for `sieve` binds `sieve`'s type to a set of tuple-types shown in figure 9; we partition the set based on the functions in which the tuple operations occur.

We first give an informal description of how an optimizer might use the information generated by the type inference system to construct a representation for tuple-space and then present a more precise description of the process.

Based on the structure of the tuples deposited into `sieve` we can deduce the following useful properties about this tuple-space:

i. It contains only tuples of length 2.

ii. There are two constants (`"head"` and `"tail"`) used only in the first field of a tuple.

iii. All formals appear only in the second field of any `read` tuple.

Given these properties, we can implement `sieve` as a structure with three components: two queues named `head` and `tail` and a symbol table (implemented in terms of a hashing function, trie, or any other suitable structure) that maps stream indices to their values. `Out` operations on `sieve` whose first field is a constant are implemented by appending to the appropriate queue; `in` or `rd` operations on these queue types simply remove the first element and block if the queue is empty. Reading or writing a stream element involves manipulating the symbol table: to read an element, the element's index is used as a key to select the appropriate element.

[6] By "minimal type" we mean a type that contains no extraneous type information. Thus, a tuple-space type is minimal if the removal of any component tuple-type would violate a type constraint relative to some context in which it occurs.

`make-stream`

$\mathcal{K}$	$\mathcal{C}$	$\mathcal{F}$	$\mathcal{S}$	$\mathbf{N}$
GEN	$0 \mapsto$ `"head"` $\times\ \mathbf{S}$ $0 \mapsto 1$	–	–	2
GEN	$0 \mapsto$ `"tail"` $\times\ \mathbf{S}$ $0 \mapsto 1$	–	–	2

`stream-car`

$\mathcal{K}$	$\mathcal{C}$	$\mathcal{F}$	$\mathcal{S}$	$\mathbf{N}$
READ	$0 \mapsto$ `"head"` $\times\ \mathbf{S}$	$1 \mapsto \tau_1$	–	2
READ	–	$1 \mapsto \mathbf{N}$	$0 \mapsto \tau_1$	2

`stream-cdr!`

$\mathcal{K}$	$\mathcal{C}$	$\mathcal{F}$	$\mathcal{S}$	$\mathbf{N}$
READ	$0 \mapsto$ `"head"` $\times\ \mathbf{S}$	$0 \mapsto \mathbf{N}$	–	2
GEN	$0 \mapsto$ `"head"` $\times\ \mathbf{S}$	–	$1 \mapsto \mathbf{N}$	2

`attach`

$\mathcal{K}$	$\mathcal{C}$	$\mathcal{F}$	$\mathcal{S}$	$\mathbf{N}$
READ	$0 \mapsto$ `"tail"` $\times\ \mathbf{S}$	–	$1 \mapsto \mathbf{N}$	2
GEN	–	–	$0 \mapsto \mathbf{N}$ $1 \mapsto \tau_2$	2
GEN	$0 \mapsto$ `"tail"` $\times\ \mathbf{S}$	–	$1 \mapsto \mathbf{N}$	2

Figure 9: The type structure of variable `sieve`.

3.6.1 A Type Assignment Algorithm

Our type assignment algorithm $\mathcal{M}$ is based on algorithm $\mathcal{W}$ given in [8]. The procedure takes as its argument a type environment A and an expression e and returns as its result a type substitution S (*i.e.*, a map from type variables to types) and a type τ such that $A \bullet\ S\ \vdash\ e : \tau$.

To define $\mathcal{M}$ we use the unification algorithm defined by Robinson [19]:

Proposition 1 There is an algorithm U which, given a pair of types, either returns a type substitution V or fails such that:

i. if $U(\tau_1, \tau_2)$ returns V, then $V\tau_1 = V\tau_2$.

ii. If S unifies τ_1 and τ_2, and $U(\tau_1, \tau_2)$ returns some V, then there is another type substitution R such that $S = RV$. Moreover, V involves only type variables in τ_1 and τ_2.

The algorithm is given for a subset of our kernel language in figure 10.

We can show in a manner similar to that described in [17] that any type yielded by $\mathcal{M}$ conforms precisely to a legal derivation produced by our inference rules given the same expression and type environment.

3.6.2 Transforming Types to Representations

In this section, we make precise the intuition given above by sketching an algorithm to generate representations for tuple-spaces given their types. The effectiveness of the algorithm lies in how cleverly it constructs a *partition* for the tuples that occupy any given tuple-space. A partition separates tuples into disjoint sets. Each set contains READ and GEN kind tuples that can potentially match with one another. The partition is constructed such that a GEN found in partition set s_1 is guaranteed *not* to match with any READ tuple found in partition set $s_j, j \neq i$. For example, tuples which have different lengths or whose corresponding fields have distinct types can never yield a non-empty substitution. A partition that contains tuples which enjoy a similar structure can be transformed into a particular data representation that takes advantage of this similarity; we describe some optimizations in this regard below.

We formalize our intuition of partitions as follows:

Definition 2 Let γ be a tuple-space type and let $\delta_1, \delta_2, \ldots, \delta_n$ be tuple-types found in γ.

A *partition* P of γ is a relation on tuple-types defined such that $(\delta_i, \delta_j) \in P$ iff

- $\mathcal{L}(\delta_i) = \mathcal{L}(\delta_j)$.
- $\Pi(\delta_i) = \Pi(\delta_j)$.

We can now define the tuple-space construction algorithm as follows:

Tuple Representation Algorithm:

Input: A tuple-space type γ containing tuple-types, $\delta_1, \delta_2, \ldots, \delta_n$.

Output: A *representation* for γ.

Method:

Algorithm $\mathcal{M}$

$\mathcal{M}(A, e) = (S, \tau)$ where:

i. if e is x and there is an assumption $x : \tau'$ in A, then $S = A$ and $\tau = \tau'$.

ii. if $e = (e_1\ e_2)$, then

$$\begin{aligned}&\text{let } \mathcal{M}(A, e_1) = (S_1, \tau_1),\\ &\quad \mathcal{M}(A \bullet S_1, e_2) = (S_2, \tau_2),\\ &\quad U(S_2(\tau_1), \tau_2 \to \beta) = V\\ &\qquad \text{(where } \beta \text{ is new)}\\ &\text{in } S = (S_1 \bullet S_2 \bullet V) \text{ and } \tau = V(\beta)\end{aligned}$$

iii. if $e = (\lambda(x)e_1)$, then

$$\begin{aligned}&\text{let } \beta \text{ be a new type variable},\\ &\quad \mathcal{M}(A[x \mapsto \beta], e_1) = (S_1, \tau_1)\\ &\text{in } S = S_1 \text{ and } \tau = S_1(\beta) \to \tau_1\end{aligned}$$

iv. if $e = (\,\texttt{make} - \texttt{ts}\,)$, then

$$\begin{aligned}&\text{let } \beta \text{ be a new type variable},\\ &\text{in } S = A[\beta \mapsto TS_\perp] \text{ and } \tau = TS_\perp\end{aligned}$$

v. if $e = (\,\texttt{out}\ e_1\ (e_2\ \ldots\ e_n))$, then

$$\begin{aligned}&\text{let } \mathcal{M}(A, e_1) = (S_1, \tau_1)\\ &\quad \mathcal{M}(A \bullet S_1, e_2) = (S_2, \tau_2)\\ &\qquad \vdots\\ &\quad \mathcal{M}(A \bullet S_{n-1}, e_n) = (S_n, \tau_n)\\ &\quad t = (e_2, \ldots, e_n)\\ &\quad [\text{GEN}, Constants(S_n, t), Formals(S_n, t), Actuals(S_n, t), n-1] = ts\\ &\text{in } S = S_n[\tau_1 \mapsto S_n(\tau_1) \cup_b ts] \text{ and } \tau = \phi\end{aligned}$$

vi. if $e = (\,\texttt{rd}\ e_1\ (e_2\ \ldots\ e_n)\ e_b)$, then

$$\begin{aligned}&\text{let } \mathcal{M}(A, e_1) = (S_1, \tau_1)\\ &\quad \mathcal{M}(A \bullet S_1, e_2) = (S_2, \tau_2)\\ &\qquad \vdots\\ &\quad \mathcal{M}(A \bullet S_{n-1}, e_n) = (S_n, \tau_n)\\ &\quad t = (e_2, \ldots, e_n)\\ &\quad [\text{READ}, Constants(S_n, t), Formals(S_n, t), Actuals(S_n, t), n-1] = ts\\ &\quad \mathcal{M}(A \bullet S_n[\tau_1 \mapsto S_n(\tau_1) \cup_b ts], e_b) = S_b, \tau_b\\ &\text{in } S = S_b \text{ and } \tau = \tau_b\end{aligned}$$

Figure 10: Type assignment algorithm for a subset of $\mathcal{TS}$.

Step 1 Partition the tuple-types found in γ.

Optimization: If the tuples to occupy a given partition are defined purely in terms of constants (*e.g.,* `(out T ("a","b"))` or `(in T ("c","d","e")` e`)`), then we can implement each constant tuple as a general semaphore. `Out` operations perform a V operation on the corresponding semaphore; `in` or `rd` expressions perform a P operation.

Step 2 Compute the set of *search keys* for each partition. A search key is an index i defined such that no tuple field in any tuple found within such a partition contains a formal at position i. The set of search keys for a given partition defines exactly those fields that are involved in the matching procedure.

Optimization: If the search keys for a given partition denote tuple indices whose contents are all constants, the partition can be implemented as an n dimensional matrix where n denotes the number of search keys for the partition. The structure of this matrix is given by allocating a dimension for each tuple index corresponding to a search key; the contents of the n^{th} dimension contains a queue to hold the binding-values for the formals found in `rd` or `in` expressions that access this partition.

For example, the partitions generated for `sieve`'s type would place tuples whose first fields are `"head"` and `"tail"` into the same partition. Since this first field is the only search key in this partition, we can build a vector of length 2 indexed by either `"head"` or `"tail"`. Each element in the vector contains a queue. `Out` operations augment the queue; `in` and `rd` operations remove an element from the front, blocking if no such element exists.

Step 3 In the general case, a partition contains a set of search keys whose corresponding tuple-field contents are not known statically. The contents of the tuple-fields represented by the search keys are used here as keys in a generalized hashing procedure. An `out` expression is implemented by hashing the contents of each search key in its tuple argument to a pair containing the tuple and the search key. An `in` or `rd` expression computes the intersection of the set of tuples returned by hashing on each of the search key arguments. If this intersection is empty, it means that no matching tuple has yet been generated; if the intersection set contains more than one element, it means that several matching tuples have been deposited. Any tuple can be chosen from this set non-deterministically.

For example, given the following expressions:

```
(out T (a,  b, c))
(rd  T (?d, f, g) E)
```

and assuming that the corresponding tuple-types reside in the same partition, we can use the second and third fields in the tuples as the search keys. To implement the `out` expression, we augment the hash structure by hashing the value of `b` to the pair:

$<$ `1, (`*value of* `a,` *value of* `b,` *value of* `c)` $>$

and the value of `c` to the pair:

$<$ `2, (`*value of* `a,` *value of* `b,` *value of* `c)` $>$

We translate the `rd` by first requiring the value of `f` and `g` to be used as hash keys to retrieve two sets of tuples. We extract from the first set all tuples belonging to pairs whose first field is `1`; we extract from the second set all tuples belonging to pairs whose first field is `2`. The intersection of these two sets gives us the set of tuples that match the template specified by the `rd` expression.

4 Conclusions

Our structural analysis technique enables us to construct a representation for first-class tuple-spaces similar to what an optimizing C.Linda compiler can generate given a more restrictive global tuple-space structure.

There is however a broad class of optimizations that still merit investigation. Our analysis has been restricted to a study of a tuple-space's structural properties; these properties are concerned with static attributes of a tuple, *e.g.,* a tuple's length, the types of its fields, etc.. Flow analysis of $\mathcal{TS}$ programs would enable us to collect information on the dynamic attributes of tuple-spaces. For example, a given `rd` expression may potentially match with n `out` expressions; however, as a result of flow analysis, we may be able to deduce that only a small number (say k) of these n `out`'s can actually be involved in any match operation with this `rd` expression. Such a situation would occur if the `rd` logically precedes (in the flow graph) the remaining $n - k$ `out` operations. Flow dependency information can be used to build a more refined partition scheme for tuple-spaces.

Time analysis is another non-standard analysis that can be profitably applied to construct efficient representations for tuple-spaces. Time analysis deals with inferring the amount of time a given expression takes to execute. In the context of tuple-space languages, time analysis can be used to answer questions of the form: given tuple expressions t_1 and t_2 found in two concurrently executing processes, will t_1 have executed before t_2 begins evaluating relative to some execution metric or scheduling policy M? Such analysis can shed light on scheduling algorithms for active tuples as well as giving us further information on representation structures for tuple-spaces.

We intend to develop these analysis in greater detail in the near future as we experiment with various implementations of the ideas detailed here.

References

[1] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data Structures for Parallel Computing. *Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[2] R. Bjornson, N. Carriero, D. Gelernter, and J. Leichter. Linda, the Portable Parallel. Technical Report RR-520, Yale Univ. Dept. of Computer Science, January 1988.

[3] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[4] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[5] Nick Carriero and David Gelernter. Applications Experience with Linda. In *Proceedings of the ACM Symposium on Parallel Programming*, pages 173–187, July 1988.

[6] Nick Carriero and David Gelernter. Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler. In *Second Workshop on Languages and Compilers for Parallelism.* MIT Press, August 1989.

[7] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixed-Points. In 4^{th} *ACM Symposium on Principles of Programming Languages*, 1977.

[8] Luis Damas and Robin Milner. Principle Type-schemes for Functional Programs. In *9^{th} ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[9] David Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of* PARLE '89*, volume 2*, pages 20–27, 1989.

[10] G. Gentzen. Investigations into Logical Deduction. In M.E.Szabo, editor, *The Collected Papers of Gerhard Gentzen.* North-Holland Press, 1969.

[11] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[12] Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, June 1989.

[13] Susanne Hupfer. Melinda: Linda with Multiple Tuple Spaces. Technical Report YALEU/DCS/RR-766, Dept. of Computer Science, Yale University, 1990.

[14] Suresh Jagannathan. Semantics and Analysis of First-Class Tuple-Spaces. Technical Report DCS/RR-783, Yale University, April 1990.

[15] Neil Jones and Alan Mycroft. Data Flow Analysis of Applicative Programs Using Minimal Function Graphs. In 13^{th} *ACM Symposium on Principles of Programming Languages Conf.*, pages 296–306, 1986.

[16] Wm Leler. Linda Meets Unix. *Computer*, 23(2):43–55, February 1990.

[17] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[18] Rishiyur Nikhil. ID Reference Manual (Version 88.0). Technical report, MIT, 1988. Computation Structures Group Technical Report.

[19] John A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12:23 –44, 1965.

[20] Ehud Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, 19(8):44–60, August 1986.