Characterizing the Paralation Model using Dynamic Assignment

Eric T. Freeman[1*] and Daniel P. Friedman[2]

[1] Yale University, Department of Computer Science, New Haven, Connecticut 06520
[2] Indiana University, Computer Science Department, Bloomington, Indiana 47405

Abstract. Collection-oriented languages provide high-level constructs for describing computations over collections. These languages are becoming increasingly popular with the advent of massively parallel SIMD machines. Even on serial machines, collection-oriented languages are interesting because they allow the user to describe computation in a concise manner. In addition, these languages can easily be compiled on superpipelined, superscalar, and vector machines because they are so rich in simple loops. In this paper we implement Sabot's paralation model [7], a collection-oriented language extension for expressing elementwise computation and explicit communication. We extend the Scheme programming language, a dialect of Lisp with lexical scope and first-class procedures, basing our implementation on Sabot and Blelloch's original code written in Common Lisp. We have taken particular care to remove all unnecessary side-effects. This re-packaging has taken two forms. First, many of the side-effects can be treated as a kind of dynamic assignment. Some others can be removed by coding in a more functional style. The remaining category of side-effects are for site-specific operations. This re-packaging has led to a surprising result: the paralation model can be described with just a single language form and a single primitive.

1 Introduction

All programming languages allow the application of procedures over collections of data. Low-level languages accomplish this by applying an iteration operator such as a for-loop over an array of data. Other languages like Lisp provide this capability at a higher level through mapping. Mapping successively applies a procedure to each item in a collection. *Collection-oriented languages* provide even higher-level operations that manipulate collections as a whole [1, 2]. Typical operations might include adding a constant to an entire collection, reducing a collection to the sum of its elements, or removing all even elements of a collection.

Collection-oriented languages are not new, APL provides many high-level operations for manipulating arrays and matrices. Only now are these languages gaining popularity because of their natural mapping to highly parallel machines. These machines are appropriate for collection-oriented computation because they can perform operations in parallel over large data structures. As a result a new programming paradigm (data-parallel algorithms) has shown that a large class of problems have inherently parallel solutions and programmers benefit from being able to think about

* Research done while author was at Indiana University.

solving problems in a parallel manner[6]. It is this facet that interests us—the increase in expressibility that occurs when we add collection-oriented extensions to a language.

Sabot's paralation model [7] is a high-level, abstract language extension for describing concurrent operations over collections of objects. These extensions are supported through a data type, the *field*, one special form, **elwise**, and a communication primitive, **create-open-mapping**[3]. In this paper we extend the Scheme programming language (see for example [4]), basing our implementation on Sabot and Blelloch's original extensions to Common Lisp. We have chosen Scheme for this task because of its lexical scope, first-class procedures, and language extension facilities.

This paper is descriptive in nature. We do not attempt to augment the paralation model but rather to re-package it to better understand its operation. Where possible, we have minimized the use of side-effects and introduced fluids to achieve this goal.

We review the field data type along with the concept of a paralation. Next we present **elwise**, show examples of its use, and give its implementation, which involves an interesting use of fluid assignment with mapping. Similar discussion of the **create-open-mapping** primitive, an operator that allows explicit communication between collections, follows.

2 Paralations and Fields

The table below depicts a relation R with column headings that represent domains. For instance, the domain **Fruit** consists of all possible fruit names, the domain **Quantity** consists of the number of each item in inventory, etc. Each row is called a tuple and has one value from each domain. $\#R$, the cardinality of the relation, is the number of tuples.

Inventory Number	Fruit	Quantity	Price
0	apple	2	.25
1	kiwi	5	.99
2	tomato	11	.75
3	pear	12	.60
4	orange	8	.55
5	grape	20	.89

The paralation model is based on this relational approach. A *paralation* (parallel relation) is like a relation except that all values in each tuple (row) are considered *elementwise*. If values are elementwise, computation can be performed on them in parallel, or at least expressed in a concise manner.

The paralation model calls each column a field; fields are data objects that contain one value for each tuple. Tuples in the model reside at a specific *site*. Although these sites don't necessarily have an ordering, they are enumerated by an *index field*. If we

[3] This is our characterization of the paralation model. Sabot's model contained three primitives: **elwise**, **move**, and **match**.

let R be an arbitrary paralation then its index field contains the values $0, 1, ..., \#R-1$. This acts much like a primary key in relational databases[3]. In the table, **Inventory Number** is the index field. Each field is a discrete data object, however all fields that belong to a paralation are bound together into a paralation by an association with the same index field.

We will soon introduce an operator, **elwise**, for computing in an elementwise manner, and an operator, **create-open-mapping**, for communicating between paralations.

2.1 Representations

We define a field to be an abstract data type consisting of four procedures: **field-ref**, **field-set!**, **field-index**, and **field-map**. Our abstract type is implemented through a general record facility similar to **defstruct** of Common Lisp[8]. We represent a field as a record structure containing a vector and a pointer to an index field.

(**define-record** *field* (*vector index*))

Here **define-record** defines a predicate, **field?**, and three procedures: **make-field**, **field->vector**, and **field->index**. Figure 1 displays the relationship of the fields in the table using this representation. These four fields belong to the same paralation and one of them, **Inventory Number**, is an index field. This is evident by examining the index field's pointer, which is self-referential.

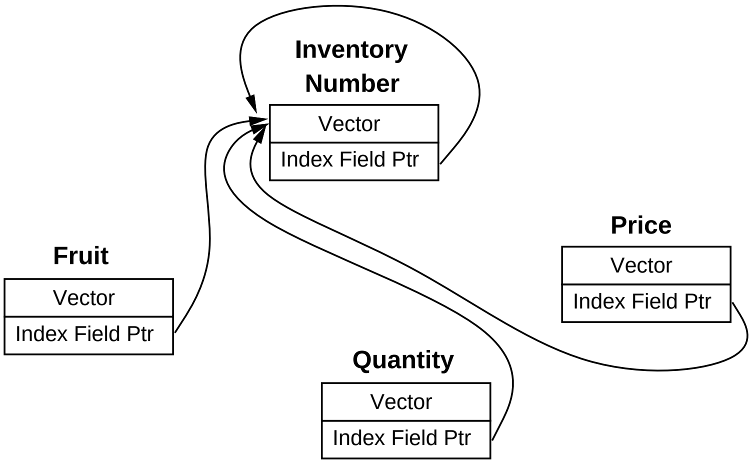

Fig. 1. Three fields in a paralation.

Below we give the implementation of the field abstract data type. **Field-ref** has two parameters, a *field* and an integer *index*, and returns the element at position *index* in *field*. **Field-set!** takes the same parameters as **field-ref** plus a *value* and assigns *field* at position *index* to *value*. **Field-index** (not to be confused with **field->index**) takes any field and returns its index field (the implementation of this procedure will become apparent in the next section). **Field-map** returns a field that is the result of applying a procedure to each element of *field*.

```
(define field-ref
  (lambda (field index)
    (vector-ref (field->vector field) index)))

(define field-set!
  (lambda (field index value)
    (vector-set! (field->vector field) index value)))

(define field-index
  (lambda (field)
    ((field->index field))))

(define field-map
  (lambda (proc field)
    (let ((l (vector->list (field->vector field))))
      (make-field (list->vector (map proc l)) (field->index field)))))
```

Field ADT

2.2 Paralation Construction

To create a paralation, we first make an index field and then use the index field as a prototype for creating other fields in the paralation. We now define a procedure **make-paralation** that creates index fields.

```
(define make-paralation
  (lambda (length)
    (letrec ((index-field
               (make-field (list->vector (iota length))
                           (lambda () index-field))))
      index-field)))
```

Make-paralation takes one parameter, the length of the paralation, from which it creates and returns an index field. Stepping through the evaluation of **make-paralation**, first we create a field record, supplying a vector and a self-referencing thunk. The vector holds the sequence of integers that enumerates the sites of the paralation. We create this vector through **iota**, a procedure that returns a list with elements from 0 to $length - 1$ (e.g., (**iota** 3)⇒(0 1 2)); **list->vector** then transforms this into the vector we need. The thunk, when invoked, returns the index field, which in this case is a self reference to the field we are creating. We then return the record. For example,

(**make-paralation** 5) $\Rightarrow$ #F(0 1 2 3 4)

Here we create a paralation of length five and an index field is returned that enumerates the sites of the paralation. This example uses a special prefix to indicate our field print representation, which displays only the contents of the vector and not the index field pointer.

3 Elwise

Elwise is a special form for expressing elementwise computation. With **elwise** we can apply any Scheme expression to an entire paralation; the formal parameters of the **elwise** form specify which identifiers are to be treated elementwise within a body of code. We now give the syntax for **elwise**.

(**elwise** (*id1 id2* ...) *exp1 exp2* ...)

The **elwise** form is syntactically similar to **lambda**, but semantically it behaves more like a **let**. **Elwise** consists of a list of one or more identifiers, called formal parameters, followed by a sequence of expressions called the body. **Elwise**, however, binds its formal parameters in a different way: when a **lambda** is applied it evaluates its arguments, binds them to the formal parameters, and evaluates the body in the scope of these new bindings. **Elwise** requires that its formal parameters already be bound in the scope of the **elwise** expression. In addition, these formal parameters must be bound to fields that belong to the same paralation.

The **elwise** form operates by repeatedly applying a body over each site of a paralation. The result is a new field where the value at each site is obtained from the evaluation of the body over that site. Consider the following:

fruit $\Rightarrow$ #F(apple kiwi tomato pear orange grape)

(**cons** *fruit* '(juice)) $\Rightarrow$ (#F(apple kiwi tomato pear orange grape) juice)

Here *fruit* is treated as a normal Scheme object and **cons**'d to the list (juice). With the **elwise** form we can perform this computation treating *fruit* in an elementwise manner, applying **cons** over each site.

(**elwise** (*fruit*) (**cons** *fruit* '(juice))) $\Rightarrow$

#F(
(apple juice)
(kiwi juice)
(tomato juice)
(pear juice)
(orange juice)
(grape juice))

In this example the value at each site is **cons**'d onto '(juice). We now give a few more examples of **elwise** using two fields:

inventory-number ⇒ #F(0 1 2 3 4 5)

quantity ⇒ #F(2 5 11 12 8 20)

(**elwise** (*inventory-number quantity*) (**cons** *inventory-number quantity*))

⇒ #F((0 . 2) (1 . 5) (2 . 11) (3 . 12) (4 . 8) (5 . 20))

(**elwise** (*quantity*) (**cons** *inventory-number quantity*))

⇒
#F(
(#F(0 1 2 3 4 5) . 2)
(#F(0 1 2 3 4 5) . 5)
(#F(0 1 2 3 4 5) . 11)
(#F(0 1 2 3 4 5) . 12)
(#F(0 1 2 3 4 5) . 8)
(#F(0 1 2 3 4 5) . 20))

In the first example, *inventory-number* and *quantity* are elementwise **cons**'d together. In the second example, *quantity* is treated elementwise in the computation whereas *inventory-number* is not. The result is a new field where at each site the field *inventory-number* has been **cons**'d onto the value of *quantity*.

3.1 Implementation

The **elwise** special form is implemented through a syntactic extension facility[4]. This facility, **extend-syntax**, expands input expressions into output expressions based on pattern matching. In this section we first introduce **extend-syntax** through an implementation of the Scheme form **let**. We then implement a related form **fluid-let** that is subsequently modified to implement **fluid-let-at-site**. Last we implement **elwise** combining **field-map**, which we have already introduced, and **fluid-let-at-site**. The reader who is familiar with **extend-syntax** and **fluid-let** should skip to section 3.4. The implementations of **let** and **fluid-let** are for the purpose of explaining **fluid-let-at-site** and are not used in the rest of the paper.

3.2 Let

Let is a syntactic form, derived from **lambda**, that creates local bindings and then evaluates one or more expressions (a body) in the scope of the new bindings. For example, in the expression (**let** ((a 1) (b 2)) (+ a b)), a is bound to 1, b is bound to 2, and (+ a b) is evaluated in the scope of those bindings. **Let** expressions can be rewritten in terms of **lambda**; the previous **let** expression can be rewritten as ((**lambda** (a b) (+ a b)) 1 2). We now define an expansion for **let** with **extend-syntax**.

```
(extend-syntax (let)
  ((let ((var val) ...) e1 e2 ...)
   ((lambda (var ...) e1 e2 ...) val ...)))
```

In our **let** example, any expression beginning with **let**, followed by zero or more pairs of *vars* and *vals* and then a body, is matched against the pattern

(**let** ((*var val*) ...) *e1 e2* ...)

The expression is then expanded into a **lambda** expression created by listing the *vars* as formals and *e1 e2* ... as the body. The **lambda** expression is packaged in an application with *val* ... as operands.

Extend-syntax begins with a *keyword* followed by a *pattern/expansion* pair. Whenever an expression containing *keyword* is encountered, it is matched against *pattern* and expanded according to *expansion*. The ellipsis notation specifies that a variable in *pattern* can match zero or more expressions in the input. Ellipses in *expansion* are expanded accordingly.

3.3 Fluid-let

Fluid-let assigns existing identifiers to new values, executes a body of code, and then reassigns the identifiers to their original values. Unlike **let**, **fluid-let** does not determine the scope of identifiers. Instead it establishes a new value for them. For example, the following expression demonstrates the difference between **let** and **fluid-let**.

```
(let ((a 3))
  (let ((proc (lambda () (+ a 5))))
    (list
      (fluid-let ((a 7))
        (proc))
      a
      (let ((a 7))
        (proc)))))

⇒ (12 3 8)
```

We now present the form **fluid-let**, which uses **with** [4].

```
(extend-syntax (fluid-let)
  ((fluid-let ((id val) ...) e1 e2 ...)
   (with (((t ...) (map (lambda (x) (gensym)) '(id ...))))
     (let ((body (lambda () e1 e2 ...))
           (swap (let ((t val) ...)
                   (lambda ()
                     (let ((temp t))
                       (set! t id)
                       (set! id temp)) ...))))
       (dynamic-wind swap body swap)))))
```

The **with** form is used in **extend-syntax** and is evaluated before expansion; similar to a **let**, **with** is followed by binding/expression pairs. Each expression is evaluated at expansion time and the result bound to a symbol or pattern. The body of the **with** expression is then expanded. Here the **with** generates a unique symbol for each *id* in the **fluid-let** and binds these to the pattern *t* ... This is accomplished through mapping the **lambda** expression over each *id*. The lambda expression takes an *id* and returns a new symbol through the procedure **gensym**, which creates unique symbols.

After unique identifiers are created for all *ids*, we expand the body of the **with**. The **let** expression first assigns *body* to a procedure that evaluates the expressions *e1 e2* ... Next *swap* is bound to a procedure that exchanges the values in each *id* and temporary *t*. The body of the **let** contains a call to **dynamic-wind**. The procedure **dynamic-wind** takes three thunks as arguments, invokes all three in order, and returns the value of the second thunk. In the event that an error occurs in the evaluation of the second thunk, **dynamic-wind** guarantees that the third thunk will always be evaluated (in the presence of first-class continuations, **dynamic-wind** is actually more complicated [5].).

The evaluation of **fluid-let** occurs as follows: first, the temporary variables, *t* ..., are assigned to the new values, *val* ... Next, we enter the **dynamic-wind** and it evaluates *swap* to exchange the values of the temporaries and the *ids*. Then *body* is evaluated. Last, **dynamic-wind** again evaluates *swap* to exchange the values of the temporaries and the *ids* so that the *ids* acquire their original values. **Dynamic-wind** then returns the result of evaluating *body*.

3.4 Fluid-let-at-site

We now present a variant of **fluid-let** called **fluid-let-at-site**. This new form has the following syntax.

(**fluid-let-at-site** (*id1 id2* ...) *e1 e2* ...)

Fluid-let-at-site takes a list of identifiers that are bound to fields and a body of expressions as parameters and returns a procedure of one argument, a site, that when applied has the effect of evaluating a body of code over one site of all fields referenced by an *id*. For example, consider

$x \Rightarrow$ #F(51 41 31 21 11)

$y \Rightarrow$ #F(12 32 52 72 92)

((**fluid-let-at-site** (*x y*) (+ *x y*)) 2)

The body (+ *x y*) is evaluated over site 2 (where x is 31 and y is 52) and the result is $31 + 52 = 83$. With an argument of site 3, the result is $21 + 72 = 93$, etc.

Below we implement **fluid-let-at-site**. Like **fluid-let**, we create a temporary variable for each *id*, binding each temporary to the value of *id* (a field) and we create a binding for *body* by wrapping its expressions in a thunk. Two procedures are then defined, *swap-in* and *swap-out*. The procedure *swap-in* assigns each *id* to the value of the field *id* at *site*. The procedure *swap-out* first copies the value of *id* back into the field stored in the temporary variable and then reassigns each *id* to its original value. A procedure of one argument, a site, is returned. When evaluated *swap-in*, *body*, and *swap-out* are evaluated within **dynamic-wind**.

```
(extend-syntax (fluid-let-at-site)
  ((fluid-let-at-site (id ...) e1 e2 ...)
   (with (((t ...) (map (lambda (x) (gensym)) '(id ...))))
     (let ((t id) ...
           (body (lambda () e1 e2 ...)))
       (let ((swap-in
               (lambda (site)
                 (lambda ()
                   (set! id (field-ref t site)) ...)))
             (swap-out
               (lambda (site)
                 (lambda ()
                   (field-set! t site id) ...
                   (set! id t) ...))))
         (lambda (site)
           (dynamic-wind (swap-in site) body (swap-out site))))))))
```

At first it might not be clear why in *swap-out* the values of the *id*s need to be copied back into each field at *site*—this is necessary when side-effects are used. For example, consider the following **fluid-let-at-site** expression.

(**fluid-let-at-site** (*x*) (**set!** *x* 1) (+ *x* 1))

When this expression is applied to an arbitrary site, it changes the value of x to 1 and then evaluates (+ *x* 1) to obtain a value of 2. After the evaluation of this expression, the value of x should be mutated to 1. If we follow the evaluation of the **fluid-let-at-site** form, x is bound to the value of the field x at *site*. The side-effect occurs, changing x to a value of 1 and then the body is evaluated. If we did not copy the value of x back into the field at this point, then the side-effect would not manifest itself in the field x. It would be lost: although the expression would evaluate to the right value, the field x would not contain the correct element at *site*.

3.5 Implementing Elwise

Elwise is implemented by iterating **fluid-let-at-site** over each site of a paralation. To examine the way this works we give an example of an **elwise** expression and its expansion.

(**elwise** (x y) (+ x (− y 3)))

expands into

(**field-map**
  (**fluid-let-at-site** (x y) (+ x (− y 3)))
  (**field-index** x))

Field-map, defined in the field ADT, returns a field that is the result of repeatedly applying a procedure to each site of a field. In the expansion of **elwise**, the procedure is created by **fluid-let-at-site** and it is applied to the index field of the formal parameters. We retrieve the index by applying **field-index** to only the first field of the formal parameters. Although a more robust implementation would check that all formal parameters are from the same paralation, here we do no error checking. In fact, with this relaxed implementation, fields from different paralations can be used in an **elwise** expression so long as they all have the same length.

We now give the **extend-syntax** implementation of **elwise**.

```
(extend-syntax (elwise)
  ((elwise (id1 id2 ...) e1 e2 ...)
   (field-map
     (fluid-let-at-site (id1 id2 ...) e1 e2 ...)
     (field-index id1))))
```

4 Communication

The paralation model performs communication between paralations through *mappings*. In this section we introduce mappings and the primitive for creating them.

4.1 Mappings

A (partial, multi-valued) mapping is a procedure that creates a new field (in the target field's paralation) by permuting and combining the elements of a data field according to a communication pattern. The communication pattern consists of one-way pointers from the paralation of the source field to the paralation of the target field. A pointer is constructed when an element in the source field is the same as (equal to) an element in the target field. Figure 2 depicts a communication pattern; notice that the mapping is partial and multi-valued. As a result it is necessary to combine some elements into a single value at some sites and supply default values

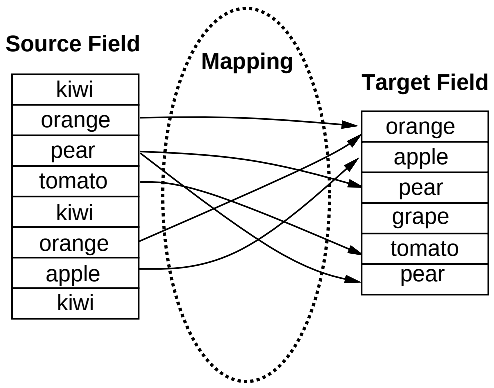

Fig. 2. A mapping.

for other sites when applying a mapping. In addition, any elements in the source field may map to two or more values in the target field, for example pear in figure 2.

The creation of a mapping is a two-step process. The **create-open-mapping** primitive creates an *open-mapping*, a procedure of two arguments: a combining procedure and a *default* field. When an open mapping is applied to a combining procedure and a default field, it returns a *closed-mapping*. A combining procedure must be a binary operator, which successively combines two elements into one. For instance if we wish to sum all values at one site (and they are numbers) then we can use + as a combining procedure. The default field is used to supply values for sites with no incoming pointers.

A closed-mapping is a procedure of one argument, a data field, that returns a new field created by mapping each element of the data field to a site in the new field. As an example:

source ⇒ #F(kiwi orange pear tomato kiwi orange apple kiwi)

target ⇒ #F(orange apple pear grape tomato pear)

default ⇒ #F(d0 d1 d2 d3 d4 d5 d6 d7)

data ⇒ #F(a b c d e f g h)

(((**create-open-mapping** *source target*) **cons** *default*) *data*)

⇒ #F((b . f) g c d3 d c)

Here we create an open mapping from *source* to *target* and then close the mapping with the combining procedure, **cons**, and *default*. We then apply this closed mapping to *data* and obtain the resulting field.

4.2 Implementation

We now implement the **create-open-mapping** primitive, where possible exploiting **elwise**. Our algorithm is based on Sabot and Blelloch's original code.

4.3 Create-open-mapping

The **create-open-mapping** primitive takes fields, *source* and *target*, and creates fields, *fwd-ptr* and *distr-ptr*, that contain a generalized form of the communications pattern. The elements of *fwd-ptr* are forward pointers mapping each location in the source paralation to a location in the target paralation. These pointers are created by finding the position of each element of *source* in *target* using **elwise** to apply the procedure **field-position** to every element of *source*. The procedure **field-position** is similar to the Common Lisp function **position**[8]. This procedure has two parameters, a value and a field, and returns the integer site of the first occurrence of the value in the field. If the value does not occur in the field, false is returned. Since we only find the first position of each element, we lose information for one-to-many mappings. This is why *distr-ptr* is needed.

```
(define create-open-mapping
  (lambda (source target)
    (let ((fwd-ptr (elwise (source)
                     (field-position source target))))
      (let ((distr-ptr (elwise (target)
                         (let ((val (field-position target source)))
                           (if val (field-ref fwd-ptr val) #f)))))
        (lambda (bin-op default)
          (let ((combiner (lambda (x y) (if (neutral? x) y (bin-op x y)))))
            (lambda (data-field)
              (let ((destination (elwise (target) neutral-value)))
                (elwise (fwd-ptr data-field)
                  (if (number? fwd-ptr)
                      (field-set! destination fwd-ptr
                        (combiner (field-ref destination fwd-ptr) data-field))))
                (elwise (distr-ptr default)
                  (if distr-ptr (field-ref destination distr-ptr) default))))))))))
```

The elements in *distr-ptr* contain pointers used to distribute the values of one-to-many mappings from the site of the first occurrence to all duplicate sites. The construction of *distr-ptr* is as follows: using **elwise**, we examine each element of *target* and assign to *val* the position of *target* in *source*. If *val* is not false, we retrieve the value of *fwd-ptr* at location *val*. Each location then contains a pointer to the first occurrence of that element, or false if there is no pointer to that location.

We compute the mapping by passing over the *fwd-ptr* to create all the initial values and then passing over *distr-ptr* to distribute the values to duplicate sites. As a result we only have to compute duplicate values once. We now give an example of generating the *fwd-ptr* and *distr-ptr* using *source* and *target* from our previous example.

fwd-ptr $\Rightarrow$ #F(#f 0 2 4 #f 0 1 #f)

distr-ptr $\Rightarrow$ #F(0 1 2 #f 4 2)

After their creation the two pointer fields are closed within a **lambda**. This **lambda** represents the open mapping and expects *bin-op* (the combining procedure) and a default field. We then create a more general form of *bin-op*, called *combiner*, as follows: if *combiner* is applied to a *neutral* value and an argument, it returns the argument, ignoring the neutral value. Otherwise *combiner* combines the two arguments with *bin-op*. The neutral value is defined as the string "neutral" (all strings are unique in Scheme). The predicate **neutral?** simply checks whether its argument is equal to the neutral value. We will see how this neutral value is used shortly. Finally, all fields are closed in a second **lambda**, the closed mapping, that is expecting *data-field*. When *data-field* is supplied, we immediately create a *destination* field that will hold the values mapped from the *data-field*. Initially we set every site in *destination* to be the *neutral* value. This neutral value tells the combining procedure that no value has yet been stored in a site.

We now follow the evaluation of the **create-open-mapping** code after a closed mapping has been applied. We compute all first occurrences of elements with the initial **elwise** (this **elwise** is used for side-effect only and the field returned is of no concern to us; we could implement an **elwise** form that returns no field). Each element of *data-field* is mapped to its new location in *destination* by examining *fwd-ptr*. If *fwd-ptr* is a number (pointer) then *data-field* maps to location *fwd-ptr* in the *destination* field. We move this value into the destination field by combining it with the value already stored there. If nothing (the neutral value) is stored there, the *combiner* returns the value of *data-field*. Otherwise, the *combiner* combines the two values using *bin-op* and returns that value. The **field-set!** actually assigns the value to the *destination* field. To continue the example, if we supply the combining procedure **cons**, the default field *default*, and the data field *data*, then after the first **elwise**, *destination* has the value #F((b . f) g c "neutral" d "neutral").

With this accomplished, we need only create the final field, distributing values to duplicate locations. This is done by examining each element, call it e, of *distr-ptr* and returning the element in *destination* at location e. In the case where *distr-ptr* is false we return the value of the *default-field*. Finally we use a second **elwise**, which treats *distr-ptr* and *data-field* elementwise and performs the operation above.

5 Conclusion

In the paralation model we express parallel computation by denoting the identifiers that are treated elementwise. Such identifiers practice a type of variable sharing—at each paralation site a different value occupies the variable. We have demonstrated that this variable sharing aspect of the model can be implemented through a variant of Scheme's dynamic assignment. Although dynamic assignment is difficult to understand, it is primarily a way to manage extent of values. In our serial implementation the extent of the values is changed over time as we successively compute over each site. On parallel machines this extent is over different processors. The point of this paper is to further refine the structure of the paralation model. In the process we show that the extension in expressibility is through a single procedure, **create-open-mapping**, and a single special form, **elwise**. The paper is primarily descriptive and is meant to create interest in this fascinating model of computation. Future work will lead in a more formal direction.

6 Acknowledgments

We gratefully acknowledge Gary Sabot for reading drafts and for answering our questions about the paralation model. In addition we thank Elisabeth Freeman, David Kaminsky, Julia Lawall, and Shinn-Der Lee for reading drafts, and Dorai Sitaram for SLaTeX. This work was supported in part by the National Aeronautics and Space Administration under Grant NGT-50589 and by the National Science Foundation under Grants CCR89-01919 and CCR90-00597.

References

1. Guy E. Blelloch: *Vector Models for Data-Parallel Computing.* The MIT Press, Cambridge, Massachusetts, 1990.
2. Guy E. Blelloch and Gary W. Sabot: Compiling Collection-Oriented Languages Onto Massively Parallel Computers. *J. of Par. and Distr. Comp. 8*, 2, 119-134.
3. C. J. Date: *An Introduction to Database Systems.* Addison-Wesley, Reading, Massachusetts, 1977.
4. R. Kent Dybvig: *The Scheme Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
5. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes: *Essentials of Programming Languages.* The MIT Press, Cambridge, Massachusetts, 1992.
6. Daniel W. Hillis, and Guy L. Steele Jr.: Data Parallel Algorithms. *Communications of the ACM 29*, 12, 1170-1183.
7. Gary W. Sabot: *The Paralation Model: Architecture-Independent Parallel Programming.* The MIT Press, Cambridge, Massachusetts, 1988.
8. Guy L. Steele Jr.: *Common LISP: The Language Second Edition.* Digital Press, 1990.