A Calculus for Exploiting Data Parallelism on Recursively Defined Data
(Preliminary Report)*

Susumu Nishimura and Atsushi Ohori

Research Institute for Mathematical Sciences
Kyoto University, Kyoto 606-01, Japan
{nisimura,ohori}@kurims.kyoto-u.ac.jp

Abstract. Array based data parallel programming can be generalized in two ways to make it an appropriate paradigm for parallel processing of general recursively defined data. The first is the introduction of a parallel evaluation mechanism for dynamically allocated recursively defined data. It achieves the effect of applying the same function to all the subterms of a given datum in parallel. The second is a new notion of recursion, which we call parallel recursion, for parallel evaluation of recursively defined data. In contrast with ordinary recursion, which only uses the final results of the recursive calls of its immediate subterms, the new recursion repeatedly transforms a recursive datum represented by a system of equations to another recursive datum by applying the same function to each of the equation simultaneously, until the final result is obtained. This mechanism exploits more parallelism and achieves significant speedup compared to the conventional parallel evaluation of recursive functions. Based on these observations, we propose a typed lambda calculus for data parallel programming and give an operational semantics that integrates the parallel evaluation mechanism and the new form of recursion in the semantic core of a typed lambda calculus. We also describe an implementation method for massively parallel multicomputers, which makes it possible to execute parallel recursion in the expected performance.

1 Introduction

Data Parallelism is recently attracting much attention as a practical basis for massively parallel computing. The central idea underlying this paradigm is to evaluate a uniform collection of data, typically arrays, in parallel by simultaneously applying the same function to each data element in the collection. The practical importance of this paradigm is twofold. Firstly, it provides the programmer an easily understandable view of a single execution stream of a program, often called SPMD programming [Kar87], by coupling parallelism with a small

* Published in *Proc. International Workshop on Theory and Practice on Parallel Programming, 1994 (LNCS 907).*

set of elimination primitives for collection data types. Due to this property, development of an efficient parallel program is relatively easier than other forms of parallel programing. Secondly, this paradigm scales up to recently emerging massively parallel distributed memory multicomputers. By allocating a processor to each data element in a collection and restricting global synchronization to the end of iteration, data parallelism can achieve reasonable performance in a massively parallel multicomputer.

Despite these promising features, the applicability of current data parallel languages is limited to those applications whose main data structures are arrays or similar structures. Most of data parallel languages have been developed by embedding a set of parallel evaluation primitives for these structures in a conventional programming language. Examples include C* [RS87], Dataparallel C [HQ91], *Lisp [Las86], and High Performance Fortran [For93]. There are several recent proposals that integrate data parallelism in a functional calculus, including Connection Machine Lisp [WS87], Paralation Lisp [Sab88], TUPLE [Yua92], DPML [HF93] and Nesl [Ble93]. However, parallelism in these proposals is still limited to simple collection data types. From this current situation, one may think that data parallelism would be inherently restricted to applications manipulating arrays. The authors believe otherwise. By generalizing the idea of data parallelism to more general data structures and integrating it into a semantic core of a typed higher-order functional calculus, it should be possible to design a general purpose parallel programming language where the programmer can enjoy the obvious benefits of parallel evaluation in writing wider range of applications. The motivation of this work is to provide a formal account for designing such a language.

There are a few proposals to generalize array based parallelism to more general data structures. Gupta [Gup92] has considered the problem of processing dynamic data structures distributed over processors in SPMD mode and proposed a mechanism to allocate and access each cell in a given dynamic data structure by generating global names of the cells. Rogers, Reppy and Hendren [RRH92] have proposed an implementation mechanism for accessing subterms residing in different processors using process migration. Main focuses of these works are, however, parallelization of existing language mechanisms; these proposal do not provide any general language construct for writing data parallel programs with recursive data.

In this paper, we attempt to develop a computational model for data parallel evaluation of dynamically allocated recursively defined data, and integrate it in the typed lambda calculus with recursive types. Different from existing proposals of high level parallel languages, which embed a particular data structure with a set of parallel primitives, the approach taken here is to extend the lambda calculus with mechanisms to *define* parallel operations for recursively defined data structures. The calculus therefore supports far more general data parallel programming in a uniform manner. In particular, it allows data parallel programming with various common data structures in functional languages such as lists and trees, which are represented as dynamically allocated recursively

defined data.

The crucial problem in achieving this goal is to find a proper mechanism for parallel application of a function to all the subterms of a given recursively defined data. In the case of an imperative language with arrays, it is sufficient to provide a simple primitive such as `FORALL (k=1:N) ... END FORALL` in High Performance Fortran to map an operation over the elements of an array. In the case of recursively defined data structures, however, finding a suitable set of parallel primitives requires more careful consideration. Since recursively defined data structures are manipulated by general recursive functions, a parallel evaluation mechanism for those data should be integrated with a form of recursive definition. In [RRH92], it was suggested that ordinary recursively defined functions can be used as units of parallel execution using speculative evaluation primitives such as `future` and `touch` of Multilisp [Hal85]. However, this strategy only exploits limited amount of parallelism, and is inadequate for data parallel programming. The problem is that a standard recursive function definition forces a series of recursive calls to be processed sequentially. For example, under this strategy, computing the sum of an integer list still requires the time proportional to the number of the elements. To achieve a proper integration of data parallelism in a functional calculus with recursive types, we should develop a new notion of recursion suitable for data parallel processing of those data.

In this paper we provide a solution to this problem and develop a typed calculus suitable for data parallel programming. We represent a recursively defined datum as a term consisting of a node constructor and a sequence of subterms, and define a mechanism to distribute each node of a recursive datum to a separate processor. A distributed recursive datum corresponds to the set of all the subterms of the datum represented by a system of equations of the form

$$\langle \alpha_1 = T_1(\alpha_2, \ldots, \alpha_n), \ldots, \alpha_i = T_i(\alpha_{i+1}, \ldots, \alpha_n), \ldots, \alpha_n = T_n \rangle$$

where α_i is a variable denoting a subterm that has the node constructor T_i and possibly contains the subterms denoted by $\alpha_{i+1}, \ldots, \alpha_n$, and α_1 denotes the root node. The usage of variables $\{\alpha_1, \ldots, \alpha_n\}$ was inspired by Gupta's global names for the cells in a dynamic data structure. We then develop a general mechanism for defining a data parallel recursive function that manipulates such a distributed datum by transforming the underlying system of equations recursively. In contrast with an ordinary recursive function, which only uses the final results of the recursive calls of its immediate subterms, the new recursion repeatedly transforms a recursive datum represented by a system of equations to another recursive datum by applying the same function to each of the equation simultaneously, until the final result is obtained. This mechanism exploits more parallelism than the speculative approach and achieves significant speedup compared to the sequential evaluation of recursive functions. We develop a calculus embodying these mechanisms, and describe an implementation method for distributed memory multicomputer.

We carry out this development in a formal framework of the typed lambda calculus with recursive types. The proposed calculus has a rigorous semantics

that accounts for parallel evaluation of recursively defined data. There are a few formal accounts for semantics of a language with data parallel construct. Data Parallel Categorical Abstract Machine [HF93] describes an execution model of data parallel programming with arrays. However, this model relies on special primitives such as `get` which exchanges data between independently executing categorical abstract machines, and these special primitives are not part of the formal framework. Suciu and Tannen [ST94] have provided a clean formal semantics for their data parallel language which contains sequences and simple parallel primitives to map functions over sequences. Our language is far more powerful than theirs, and therefore providing a formal operational semantics is more challenging. A formal semantics we have worked out for our calculus will shed some light on better understanding of general nature of data parallel programming.

This paper is a preliminary report on the development of the calculus and its implementation method. The authors intend to present more complete description of the calculus with its formal properties and the experimentation result of a prototype implementation being developed at Kyoto University. The rest of the paper is organized as follows. In Section 2, we explain the basic idea of parallel recursive function definition for recursively defined data. Section 3 defines the calculus and gives some programming examples. In Section 4, we first describe the intended execution model of the calculus in a multicomputer. We then give a parallel operational semantics in the style of natural semantics [Kah87]. Section 5 describes implementation strategies. Finally, in Section 6, we discuss some future work and conclude the paper.

2 Parallel Recursion for Recursively Defined Data

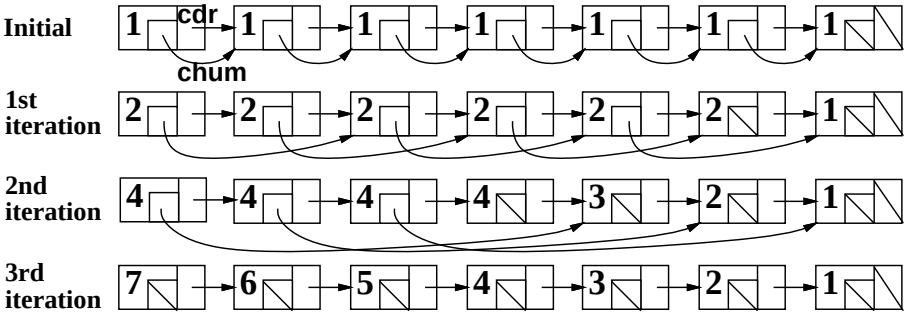
To analyze the desirable properties of data parallel function for recursively defined data, let us consider the following simple function which computes the sum of a given integer list:

```
(defun sum(L) (if (null L) 0 (+ (car L) (sum (cdr L)))))
```

The semantics of this recursive definition implies that the application of `sum` to an n element list results in a sequence of n recursive calls of `sum`, and therefore straightforward parallel evaluation requires a series of n communications that must be processed sequentially. However, Hillis and Steele [HS86] have shown that there is a data parallel algorithm to compute the sum of an n element integer list in $O(\log n)$ parallel steps. The main idea in the algorithm is to use an extra `chum` pointer to maintain information about the necessary value to complete a partially computed recursive call. The following program describes such an algorithm that computes the suffix sum of an integer list with extra `chum` pointer, where `value` and `next` pointer represent `car` and `cdr` in Lisp respectively.

```
for all k parallel do
  chum[k] := next[k]
  while chum[k] ≠ nil
    value[k]:= value[k] + value[chum[k]]
    chum[k]:=chum[chum[k]] od
```

The following figure illustrates how the suffix sum of a list is computed by the algorithm.



The above analysis suggests that there should be a new form of recursion that is to be formalized for parallel evaluation of recursively defined data other than ordinary recursion, which appears to contain inherently sequential nature. The new recursive construct proposed in the present paper is based on the observation that the techniques of Hillis and Steele can be regarded as a special case of more general form of recursion suitable for data parallel evaluation of recursively defined data. We present below the main idea using lists as an example.

The type of integer list is represented by the recursive type $\mu t.unit + int \times t$ where $unit$ is a trivial type whose only element is $*$ (which in this case is used to represent the empty list or Nil), and $+$ and $\times$ are disjoint union and product constructor, respectively. A recursive function f of type $(\mu t.unit + int \times t) \rightarrow \sigma$ in general has the following structure:

$$\text{fix } f.\lambda x.\text{case } x \text{ of } inl(*) \Rightarrow c,\ inr((h,t)) \Rightarrow S\ h\ (f\ t)$$

where c is some constant and S is some function of type $int \rightarrow \sigma \rightarrow \sigma$. In the case of `sum`, $c = 0$ and $S = \lambda x.\lambda y.x + y$. Let L be an n element list. Also let L_k be the kth sublist of L with $L_1 = L$, and x_k be the first element in the list L_k. Then the application of f to L results in the following sequence of applications:

$$f\ L_1 = S\ x_1\ (f\ L_2),\ f\ L_2 = S\ x_2\ (f\ L_3),\ \cdots,\ f\ L_n = S\ x_n\ (f\ L_{n+1}),\ f\ L_{n+1} = c$$

Let S_i be the partially applied function $S\ x_i$, and for any $i \leq j$ let $S_{i,j}$ be the composition $S_i \circ S_{i+1} \circ \cdots \circ S_{j-1} \circ S_j$. By considering each application $f\ L_i$ as an unknown value α_i, the above calling sequence can be considered as a system of equations over the set of unknowns $\alpha_1, \cdots, \alpha_{n+1}$. We can then compute the result $f\ L = \alpha_1$ by repeatedly transforming the system of equations itself in the following way:

$$
\begin{array}{rllll}
(initial) & \alpha_1 = S_{1,1}(\alpha_2) & \alpha_2 = S_{2,2}(\alpha_3) & \cdots \alpha_n = S_{n,n}(\alpha_{n+1}) & \alpha_{n+1} = c \\
(1st) & \alpha_1 = S_{1,2}(\alpha_3) & \alpha_2 = S_{2,3}(\alpha_4) & \cdots \alpha_n = S_{n,n}(c) & \alpha_{n+1} = c \\
(2nd) & \alpha_1 = S_{1,4}(\alpha_5) & \alpha_2 = S_{2,5}(\alpha_6) & \cdots \alpha_n = S_{n,n}(c) & \alpha_{n+1} = c \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
(\log n\, th) & \alpha_1 = S_{1,n}(c) & \alpha_2 = S_{2,n}(c) & \cdots \alpha_n = S_{n,n}(c) & \alpha_{n+1} = c
\end{array}
$$

Furthermore, if the composition $S_{i,j} \circ S_{j+1,k}$ is computed in constant time resulting in a function $S_{i,k}$ whose application is computed in constant time, then the above method is expected to compute the application $f\ L$ in $O(\log n)$ time. Special pointers used in Hillis and Steele's algorithms, `chum` in the above example, can be regarded as a representation of unknown variables α_i.

We call the parallel computation strategy explained above *parallel recursion*. Due to its uniformity, parallel recursion can be applied to various recursively defined data types, contributing to significant speedup in parallel programming. Furthermore, it has the conceptual generality that will serve as a basic tool for designing a parallel algorithm for recursively defined data.

To integrate parallel recursion in the typed lambda calculus, we introduce a recursive type $\mu t.\tau(t)$ as a data type whose value is represented by a system of equations of the form $\alpha_i = T(\alpha_{i+1}, \ldots, \alpha_k)$ we have explained, and a parallel function definition mechanism for these distributed recursively defined data. The basic idea is to apply a function that transforms a data constructor ($T(\_, \ldots, \_)$ above) to each equation and obtain a new system of equations. This mechanism is introduced as a term constructor that lifts a constructor transformer function of type $\tau(r) \rightarrow \sigma(s)$ to a function of type $\mu t.\tau(t) \rightarrow \mu t.\sigma(t)$. In the following sections, we will formally define the calculus where these constructs are available and give its parallel operational semantics that accounts for data parallel evaluation of recursive types.

3 Data Parallel λ-Calculus

We first define the calculus by giving its syntax and the type system. We then explain the intended meaning of the data parallel constructs introduced in this calculus and gives some programming examples. The detailed description of their semantics will be given later in section 4.

3.1 Definition of the Calculus

The set of types of the calculus is given by the following grammar:

$$\tau ::= b \mid unit \mid t \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau + \tau \mid \mu t.\tau$$

b stands for base types. t stands for type variables, and $\mu t.\tau(t)$ for recursive types. We write $\tau(t)$ to represent a type which possibly includes a free type variable t, and $\tau(\sigma)$ for the type obtained from $\tau(t)$ by substituting all the free occurrences of t with σ. As will become clear later, type variables are also used to denote identifiers of the nodes in a recursively defined datum.

The set of expressions of the calculus is given by the following syntax:

constant $\Gamma \vdash c : b$

unit $\Gamma \vdash * : unit$

identifier
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

λ-abstraction
$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \to \tau_2}$$

application
$$\frac{\Gamma \vdash M : \tau_1 \to \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}$$

fix
$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \text{fix } f(x).M : \tau_1 \to \tau_2}$$

product
$$\frac{\Gamma \vdash M : \tau_1 \quad \Gamma \vdash N : \tau_2}{\Gamma \vdash (M, N) : \tau_1 \times \tau_2}$$

fst
$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash fst(M) : \tau_1}$$

snd
$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash snd(M) : \tau_2}$$

inl
$$\frac{\Gamma \vdash M : \tau_1}{\Gamma \vdash inl(M) : \tau_1 + \tau_2}$$

inr
$$\frac{\Gamma \vdash M : \tau_2}{\Gamma \vdash inr(M) : \tau_1 + \tau_2}$$

case
$$\frac{\Gamma \vdash M : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash N_1 : \tau \quad \Gamma, y : \tau_2 \vdash N_2 : \tau}{\Gamma \vdash \text{case } M \text{ of } inl(x) \Rightarrow N_1, inr(y) \Rightarrow N_2 : \tau}$$

up
$$\frac{\Gamma, x_1 : s, \cdots, x_k : s \vdash M : \tau(s) \quad \Gamma \vdash N_i : \mu t.\tau(t) \text{ for } 1 \le i \le k}{\Gamma \vdash up\ M\ with\ x_1 = N_1, \cdots, x_k = N_k\ end : \mu t.\tau(t)}$$
(s is a new type variable.)

dn
$$\frac{\Gamma \vdash M : \mu t.\tau(t)}{\Gamma \vdash dn(M) : \tau(\mu t.\tau(t))}$$

μ-expression
$$\frac{\Gamma, f : r \to s, d : r \to \tau(r), x : \tau(r) \vdash M : \sigma(s)}{\Gamma \vdash \mu(f, d)(x).M : \mu t.\tau(t) \to \mu t.\sigma(t)}$$
(r and s are new type variables, and $\sigma(s)$ does not contain r.)

Table 1. Typing rules

$$\begin{aligned} M ::= {} & * \mid c \mid x \mid \lambda x.M \mid MM \mid \text{fix } f(x).M \mid (M, M) \mid fst(M) \mid snd(M) \\ & \mid inl(M) \mid inr(M) \mid \text{case } M \text{ of } inl(x) \Rightarrow M, inr(x) \Rightarrow M \\ & \mid up\ M\ with\ x_1 = M_1, \ldots, x_k = M_k\ end \mid dn(M) \\ & \mid \mu(f, d)(x).M. \end{aligned}$$

The type system for these expressions is defined by a set of rules to derive a typing of the form:

$\Gamma \vdash M : \tau$,

where Γ is a type assignment, which is a mapping from a finite set of variables to types. We write $\Gamma, x : \tau$ for the type assignment Γ' such that $domain(\Gamma') = domain(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ for any $y \in domain(\Gamma), y \neq x$. The set of typing rules of the calculus is given in Table 1.

Expression constructors other than the last three are standard ones in the typed lambda calculus with unit, product and sum types. The expression constructors *up M with ... end*, $dn(M)$ and $\mu(f,d)(x).M$ are for manipulating recursively defined data distributed over processors, whose intended meaning will be explained below.

3.2 Constructors for Manipulating Recursive Data

A common method for introducing recursive types of the form $\mu t.\tau(t)$ in a typed lambda calculus is to treat them as types satisfying the isomorphism $\mu t.\tau(t) \cong \tau(\mu t.\tau(t))$ and to introduce the term constructors realizing the isomorphism. In an ordinary lambda calculus, this can be done by simply introducing mappings Up of type $\tau(\mu t.\tau(t)) \rightarrow \mu t.\tau(t)$ and $Down$ of type $\mu t.\tau(t) \rightarrow \tau(\mu t.\tau(t))$.

In order to support parallel recursion explained in the previous section, we need to refine this strategy. First, we represent a recursively defined datum as a system of equations by a sequence of equations $< \alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k >$ to model a datum distributed over processors. In each equation, α_i is an identifier, called a *communication variable,* representing an individual subterm of a given recursively defined datum, and V_i is a value containing communication variables and corresponds to the constructor of the subterm identified by α_i. We further assume that an association sequence is ordered so that α_1 denotes the root node and that if α_i appears in V_j then $i > j$. We call such a sequence of equation an *association sequence.* The intention is that a separate processor is allocated for each node V_i and communication variables are used to link the nodes together.

The constructor *up M with* $x_1 = M_1, \ldots, x_k = M_k$ *end* creates an association sequence. This is an expression of type $\mu t.\tau(t)$ if $M_1, \ldots, M_k$ are expressions of type $\mu t.\tau(t)$, and M is an expression of type $\tau(t)$ under the assumption that $x_1, \ldots, x_k$ are variables of type t, where t represents immediate subterms of the recursive datum. This expression is equivalent to $Up(M[M_1/x_1, \ldots, M_k/x_k])$ in a standard representation explained above. In order to construct a proper association sequence for data parallel manipulation, however, it is essential to name the subterms $M_1, \ldots, M_k$ of type $\mu t.\tau(t)$ in M explicitly. Suppose that each M_i is evaluated to an association sequence of the form $< \alpha_1^i \rightarrow V_1^i, \cdots, \alpha_k^i \rightarrow V_k^i >$. The environment is augmented so that each x_i is bound to α_1^i, which is a "pointer" to the corresponding subterm, and then M is evaluated to V, the content of the root node of the recursive datum. V and subterms are linked together by merging all the association sequences obtained from the subterms $M_1, \ldots, M_k$ and then to add a top node $\alpha \mapsto V'$ to the merged sequence, where α is a new communication variable. The constructor *up M with* $x_1 = M_1, \ldots, x_k =$

M_k *end* denotes this operation. If $k = 0$, i.e. M is a constant not containing any subterm of type $\mu t.\tau(t)$, then we simply write $up(M)$ instead of *up* M *with end*.

As a simple example, the two element list $L = [1, 2]$ can be constructed as:

$$L = up\ inr((1, x_1))\ with\ x_1 = (up\ inr((2, x_2))\ with\ x_2 = up(inl(*))\ end)\ end$$

From this, the evaluator shall be able to construct an association sequence of the form:

$$< \alpha_1 \rightarrow inr((1, \alpha_2)), \alpha_2 \rightarrow inr((2, \alpha_3)), \alpha_3 \rightarrow inl(*) > .$$

dn converts a value of type $\mu t.\tau(t)$ to a value of type $\tau(\mu t.\tau(t))$. Its operational semantics is to convert an association sequence $< \alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k >$ to a value V_1' obtained from V_1 by "dereferencing pointers", i.e. by substituting each communication variable α_i in V_1 with the association sequence corresponding to α_i. For example, $dn(L)$, where L is the list defined above, is equal to $inr((1, (up\ inr((2, x_2))\ with\ x_2 = up(inl(*))\ end)))$, and is evaluated to the value: $inr((1, < \alpha_2 \rightarrow inr((2, \alpha_3)), \alpha_3 \rightarrow inl(*) >))$.

$\mu(f, d)(x).M$ is the data parallel function constructor in the calculus. This transforms a value of type $\mu t.\tau(t)$ to a value of type $\mu t.\sigma(t)$ by simultaneously applying the function specified by $(x).M$, a synonym for $\lambda x.M$, to each node in the value. $(x).M$ must be a function of type $\tau(r) \rightarrow \sigma(s)$ under the assumption that d and f are variables of type $r \rightarrow \tau(r)$ and $r \rightarrow s$ respectively. Here, r and s are fresh type variables denoting communication variables, and $\tau(r)$ and $\sigma(s)$ are types of the nodes in the recursively defined data of type $\mu t.\tau(t)$ and $\mu t.\sigma(t)$ respectively. When $\mu(f, d)(x).M$ is applied to a recursively defined datum represented as an association sequence of the form $< \alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k >$, it binds d to the function that converts each communication variable α_i to the corresponding value V_i, and binds f to the function that returns a fresh communication variable β_i for each α_i. It then applies the function $(x).M$ simultaneously to each element V_i of the datum to obtain a new value V_i'. The result is a new association sequence of the form $< \beta_1 \rightarrow V_1', \cdots, \beta_k \rightarrow V_k' >$.

By this construct, the user can write a function that transforms a recursively defined datum in parallel by simply writing a function that transforms each node in the datum. Moreover, this construct is not limited to any particular presupposed data types such as lists or sequences; it enables us to write a parallel function that transforms a value of type $\mu t.\tau(t)$ to a value of type $\mu t.\sigma(t)$ for any $\tau(\_)$ and $\sigma(\_)$. The combination of this construct and the ordinary recursion achieves the parallel recursion explained in the previous section for general recursive types.

3.3 Programming Examples

Let us show some examples for data parallel programming in this calculus. As a simple example, a data parallel version of Lisp `mapcar` function of type $(t \rightarrow s) \rightarrow list(t) \rightarrow list(s)$ is implemented as:

$$\lambda F.\lambda L.((\mu(f, d)(x).(\text{case } x \text{ of } inl(y) \Rightarrow inl(*), inr(y) \Rightarrow (F\,fst(y), f\,snd(y)))))\ L)$$

Writing a function in the bare syntax of our extended lambda calculus is rather tedious, but we can supply several useful syntactic shorthands. First, ML style data type declaration and pattern matching for recursively defined data can be safely combined. A data type declaration of the form

```
datatype int_list = Nil | int :: int_list
```

can be regarded as a recursive type of the form $\mu t.int + int \times t$ together with the bindings: `Nil` $= inl(*)$ and `::` $= \lambda(x,y).inr((x,y))$. After this, `Nil` and x`::`y can also be used as patterns for $inl(*)$ and $inr((x,y))$. Other data structures can be similarly treated. In examples below, we use ML style syntax of the form `case` e `of` pat `=>` e `|` ... `|` pat `=>` e for case statement with the above shorthand, where pat denotes a pattern. Next, in spirit of High Performance Fortran and other data parallel languages, we allow the following syntactic shorthand

`foreach` x `in` M `with` (f,d) `do` N for $((\mu(f,d)(x).N)\ M)$

We also use ML style (possibly recursive) function definition `fun` f x `=` ....

Using these shorthands, `mapcar` example can be written in the following more intuitive syntax:

```
fun mapcar F L =
    foreach x in L with (f,d) do
       case x of Nil => Nil | hd::tl => (F hd)::(f tl)
```

The data parallel algorithm to compute suffix sum of an integer list given in Section 2 can be expressed as in Figure 1.

Data parallel programs for the other recursive data structures can be plainly expressed in the calculus. For example, Rytter [Ryt85] has given an algorithm "parallel pebble game" that computes the suffix sum of an n node integer binary tree in $O(\log n)$ parallel steps. Such an algorithm can easily be written in our calculus. As seen from the SPMD execution model and the semantics of the calculus we shall give in the next section, the time required to execute a μ-application is roughly equivalent to one parallel step in conventional data parallel languages such as Dataparallel C [HQ91]. The calculus therefore achieves the desired goal of extending data parallelism to general recursive data.

4 Semantics of the Calculus

We first give an SPMD execution model of the calculus by describing how parallel primitives are evaluated in multicomputers. This will serve as a basic strategy to implement the calculus. We then give a formal operational semantics of the entire calculus that accounts for data parallel computation on recursive types.

4.1 An SPMD Execution Model

For the purpose of describing the execution model, we assume that the underlying hardware system is a distributed memory multicomputer consisting

```
fun suffix_sum L =
let fun mk_chum X =
        foreach x in X with (f,d) do
           case x of Nil => Nil | hd::tl => (hd,f tl)::(f tl)
    fun square X =
        foreach x in X with (f,d) do
           case x of Nil => Nil
              | (n,y)::z =>
                   (case (d y) of Nil => (n,f y)::(f z)
                       | (m,v)::w => (n+m,f v)::(f z))
    fun strip X = foreach x in X with (f,d) do
                     case x of Nil => Nil | (n,y)::z => n::(f z)
    fun scan x =
        case dn(x) of Nil => Nil
               | (n,y)::z => (case y of Nil => x
                                     | (m,v)::w => scan (square x))
in
    strip (scan (mk_chum L))
end
```

Fig. 1. Data Parallel Suffix Sum Program

of unbounded number of processors, each of which is uniquely identified by its processor id, and the system provides mechanisms for broadcasting and for inter-processor communication between any pair of processors.

The intended execution model of the calculus is a data parallel computation. Each processor executes the same program on its own copy of the data and yields the same memory state, *except for* recursively defined data created by special *up* primitive. As explained in the previous section, a recursively defined datum is modeled by an association sequence whose elements are distributed over processors, and is treated specially with the parallel constructs of the calculus.

In the multicomputer, we represent an association sequence in each processor P as a *global pointer* G_P, which is a pair of a processor id and a pointer to a local datum. For an association sequence $<\alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k>$, every processor P in the system allocates a global pointer $G_P = (P_r, p_P)$. The global pointer in every processor has, as its first element, the same processor id P_r where the root node V_1 is allocated. If a node V_i is allocated to processor P then p_P is set to the local address of V_i. If no node is allocated to P then p_P is set to a special value *Undefined*. For example, a list $[1, 2]$ is represented as shown in Figure 2, where the special value *Undefined* is represented by the box with backslash. By this representation, we achieve simple and efficient execution of the parallel primitives for general recursive types. In what follows, we use lists to explain the parallel primitives. In Section 5 we describe a technique to implement them for general recursive data structure.

To execute *up* $inr((M, x))$ *with* $x = N$ *end*, i.e. a cons expression, every

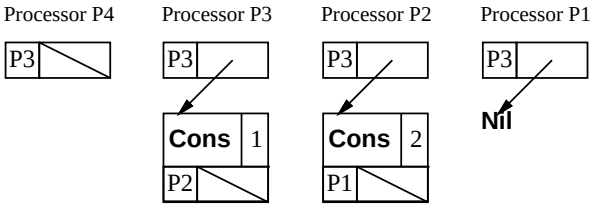


Fig. 2. Representation of a List by global pointers

processor first evaluates N to a global pointer $G_P = (P_r, p_P)$ and then evaluates $inr((M, x))$ under the environment in which x is bound to the global pointer to obtain a cons cell. Then every processor selects the same processor Q to which any elements of recursive data have not been allocated. Then the processor Q yields a global pointer (Q, q) where q is the pointer to the new cons cell. Every processor P other than Q yields a global pointer (Q, p_P). Figure 3 illustrates how $0 :: [1, 2]$ is executed and yields a global pointer in each processor.

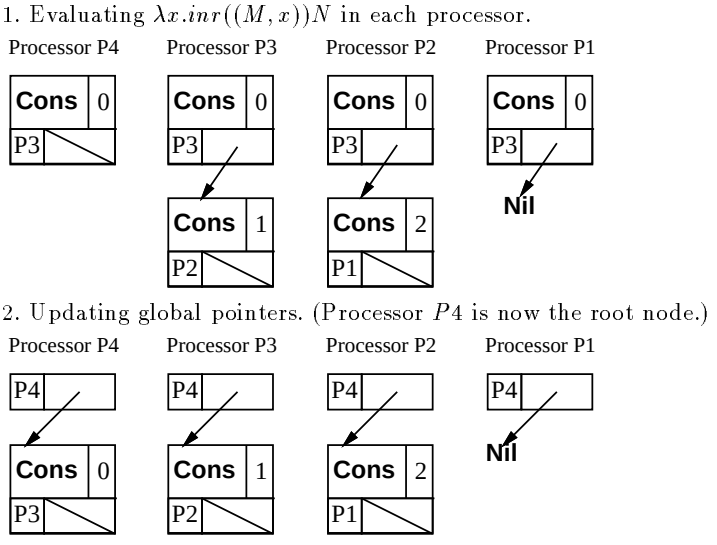


Fig. 3. Execution of *cons* on a List

$dn(M)$, where M is a distributed list, is executed as below. First, every processor P evaluates M to obtain a global pointer $G_P = (P_r, p_P)$. The processor P_r broadcasts the cons cell pointed by p_{P_r} to all the processors. Then each processor P other than P_r yields a cons cell which is obtained from the broadcasted

cell by replacing the second element of the global pointer contained in its cdr part by p_P. The processor P_r yields the cons cell pointed by p_{P_r}. Figure 4 shows the state just after $dn([1,2])$ is executed.

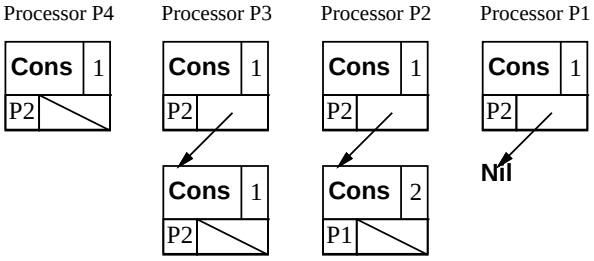


Fig. 4. Result of Applying dn to a List

Execution of an application of a μ-expression $\mu(f,d)(x).M$ is a little more complicated than the other two parallel constructs, since it includes execution of the renaming function denoted by f and the down function denoted by d. Suppose $\mu(f,d)(x).M$ is applied to a list, which is represented by a set of global pointers $G_P = (P_r, p_P)$. Every processor P in which p_P is set to *Undefined* become inactive. Then, in each remaining active processor P, the variable f is bound to a function which, given a global pointer, yields a copy of the global pointer, and the variable d is bound to a function which, given a global pointer (P,p), fetches the cons cell allocated to processor P via inter-processor communication. The variable x is bound to the local cons cell pointed by p_P, and under these bindings M is evaluated simultaneously in each processor to yield a value. The final result in each active processor is a global pointer (P_r, q_P) where q_P is a pointer to the value. Finally, the inactivated processors again become active. We show how evaluation is performed when a simple μ-expression:

$$\mu(f,d)(x).\text{case } x \text{ of } Nil \Rightarrow Nil, n :: y \Rightarrow \text{case } (d\,y) \text{ of } Nil \Rightarrow n :: (f\,y), \\ m :: z \Rightarrow (n+m) :: (f\,y)$$

is applied to a list $[1,2]$ in Figure 5, in which the boxes and arrows with dashed lines represent the old data and the old pointers respectively, the dashed arrow with letter f represent copying of global pointer, the dashed arrow with letter d represents data transfer by inter-processor communication, and the thick arrows represent data flow inside a processor.

4.2 Operational Semantics of the Calculus

An operational semantics is presented in natural semantics [Kah87].

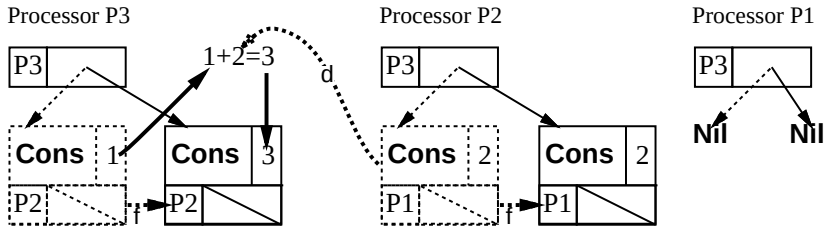


Fig. 5. μ-application on a List

Semantic Values and Environments We define the set of values (ranged over by V) for the calculus as follows:

$$\begin{aligned} V ::= {} & * \mid c \mid \mathtt{fcl}(\rho, \lambda x.M) \mid \mathtt{fxcl}(\rho, \text{fix } f(x).M) \mid \mathtt{mucl}(\rho, \mu(f,d)(x).M) \\ & \mid (V,V) \mid inl(V) \mid inr(V) \mid \alpha \mid \ <\alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k> \\ & \mid ren(\beta_1/\alpha_1, \cdots, \beta_k/\alpha_k) \mid down(<\alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k>) \mid wrong \end{aligned}$$

ρ stands for an environment, which is a mapping from variables to values. $\mathtt{fcl}(\rho, \lambda x.M)$, $\mathtt{fxcl}(\rho, \text{fix } f(x).M)$, and $\mathtt{mucl}(\rho, \mu(f,d)(x).M)$ represent the closures for function, fix, and μ-expression, respectively. (V,V), $inl(V)$, and $inr(V)$ are a pair, a left injected value, and a right injected value, respectively.

We say the occurrence of a communication variable α in V is *free*, if it is neither in an association sequence containing an association $\alpha \rightarrow V'$ for some V' nor in a renaming function of the form $ren(\cdots)$. We write $V[\alpha \backslash V']$ for the value obtained from V by replacing all free occurrences of α in V by V'.

$<\alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k>$ is an association sequence representing a recursively defined datum. It satisfies the conditions: $\alpha_1, \cdots, \alpha_k$ are all distinct, if free α_j occurs in V_i then $i < j$, and α_1 corresponds to the root node of the datum. Association sequences are often denoted by $\chi, \chi', \cdots$. The concatenation of two sequences are represented as $\chi \mathbin{@} \chi'$, and $\mathop{@}_{i=1}^{k} \chi_i$ is used to abbreviate $\chi_1 @ \cdots @ \chi_k$.

$ren(\beta_1/\alpha_1, \cdots, \beta_k/\alpha_k)$ and $down(<\alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k>)$, which are called renaming function and down function respectively, are generated and used respectively through variables f and d respectively, when an application of $\mu(f,d)(x).M$ to a distributed recursive value is evaluated.

Finally, $wrong$ represents runtime error.

Semantic Rules We define the semantics by giving a set of rules to derive a formula of the form $\rho \vdash M \Downarrow V$, representing the fact that a term M evaluates to a value V under an environment ρ. The set of rules for the calculus is given in Table 2-a and 2-b, where we omit the rules yielding $wrong$; for each rule with conditions (specified as the shape of the values), it should be understood that the complementary rule yielding $wrong$ is implicitly given.

The rules other than those for up, dn, and μ-expression are standard.

unit **constant** **identifier**

$$\rho \vdash * \Downarrow * \qquad \rho \vdash c \Downarrow c \qquad \frac{\rho(x) = V}{\rho \vdash x \Downarrow V}$$

λ-abstraction

$$\rho \vdash \lambda x.M \Downarrow \mathtt{fcl}(\rho, \lambda x.M)$$

λ-application

$$\frac{\rho \vdash M \Downarrow \mathtt{fcl}(\rho', \lambda x.M') \quad \rho \vdash N \Downarrow V' \quad \rho', x \mapsto V' \vdash M' \Downarrow V}{\rho \vdash MN \Downarrow V}$$

fix

$$\rho \vdash \text{fix } f(x).M \Downarrow \mathtt{fxcl}(\rho, \text{fix } f(x).M)$$

fix-application

$$\frac{\rho \vdash M \Downarrow \mathtt{fxcl}(\rho', \text{fix } f(x).M') \quad \rho \vdash N \Downarrow V' \quad \rho', f \mapsto \mathtt{fxcl}(\rho', \text{fix } f(x).M'), x \mapsto V' \vdash M' \Downarrow V}{\rho \vdash MN \Downarrow V}$$

up

$$\frac{\rho \vdash N_i \Downarrow < \alpha_i \rightarrow V_i > \mathtt{@}\chi_i \text{ for each } 1 \leq i \leq k \quad \rho, x_1 \mapsto \alpha_1, \cdots, x_k \mapsto \alpha_k \vdash M \Downarrow V}{\rho \vdash up(\lambda x_1 \cdots x_k.M, N_1, \cdots, N_k) \Downarrow < \alpha \rightarrow V > \mathtt{@}(\mathtt{@}_{i=1}^{k}(< \alpha_i \rightarrow V_i > \mathtt{@}\chi_i))}$$

(α is a new communication variable.)

dn

$$\frac{\rho \vdash M \Downarrow < \alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k >}{\rho \vdash dn(M) \Downarrow V_1[\alpha_2 \backslash < \alpha_2 \rightarrow V_2, \cdots, \alpha_k \rightarrow V_k >] \cdots [\alpha_k \backslash < \alpha_k \rightarrow V_k >]}$$

Table 2.-a Parallel Operational Semantics (to be continued)

The rule for *up M with* $x_1 = N_1, \cdots, x_k = N_k$ *end* first evaluates each subterm N_i, and if each of them yields an association sequence $< \alpha_i \rightarrow V_i > \mathtt{@}\chi_i$, then extracts the communication variable α_i of the first element from each of the sequences to obtain a sequence α_1, ..., α_k, and then it passes the extracted sequence to the constructor function to obtain a value V. The final result is the association sequence $< \alpha \rightarrow V > \mathtt{@}(\mathtt{@}_{i=1}^{k}(< \alpha_i \rightarrow V_1 > \mathtt{@}\chi_i))$ where α is a new communication variable. Introduction of a new communication variable corresponds to allocation of a new processor. The rule for $dn(M)$ first evaluates M, resulting an association sequence $< \alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k >$. Then, as the global pointer contained in the cdr part of the distributed cell is updated in the SPMD execution model, it returns the value obtained by instantiating its immediate subterms, i.e. by replacing each free occurrence of $\alpha_i (2 \leq i \leq k)$ in V_1 with association sequence $< \alpha_i \rightarrow V_i, \cdots, \alpha_k \rightarrow V_k >$.

pair

$$\frac{\rho \vdash M \Downarrow V \quad \rho \vdash N \Downarrow V'}{\rho \vdash (M, N) \Downarrow (V, V')}$$

fst

$$\frac{\rho \vdash M \Downarrow (V, V')}{\rho \vdash fst(M) \Downarrow V}$$

snd

$$\frac{\rho \vdash M \Downarrow (V, V')}{\rho \vdash snd(M) \Downarrow V'}$$

inl

$$\frac{\rho \vdash M \Downarrow V}{\rho \vdash inl(M) \Downarrow inl(V)}$$

inr

$$\frac{\rho \vdash M \Downarrow V}{\rho \vdash inr(M) \Downarrow inr(V)}$$

case

$$\frac{\rho \vdash M \Downarrow inl(V') \quad \rho, x \mapsto V' \vdash N_1 \Downarrow V}{\rho \vdash \text{case } M \text{ of } inl(x) \Rightarrow N_1,\ inr(y) \Rightarrow N_2 \Downarrow V}$$

$$\frac{\rho \vdash M \Downarrow inr(V') \quad \rho, y \mapsto V' \vdash N_2 \Downarrow V}{\rho \vdash \text{case } M \text{ of } inl(x) \Rightarrow N_1,\ inr(y) \Rightarrow N_2 \Downarrow V}$$

μ-expression

$$\rho \vdash \mu(f,d)(x).M \Downarrow \texttt{mucl}(\rho, \mu(f,d)(x).M)$$

μ-application

$$\frac{\begin{array}{l}\rho \vdash M \Downarrow \texttt{mucl}(\rho', \mu(f,d)(x).M') \quad \rho \vdash N \Downarrow <\alpha_1 \rightarrow V_1', \cdots, \alpha_k \rightarrow V_k'> \\ \rho',\ f \mapsto ren(\beta_1/\alpha_1, \cdots, \beta_k/\alpha_k), \\ \quad d \mapsto down(<\alpha_1 \rightarrow V_1', \cdots, \alpha_k \rightarrow V_k'>), x \mapsto V_i' \vdash M' \Downarrow V_i \quad \text{for } 1 \leq i \leq k\end{array}}{\rho \vdash MN \Downarrow <\beta_1 \rightarrow V_1, \cdots, \beta_k \rightarrow V_k>}$$

(β_1, ..., β_k are new communication variables.)

renaming

$$\frac{\rho \vdash M \Downarrow ren(\beta_1/\alpha_1, \cdots, \beta_k/\alpha_k) \quad \rho \vdash N \Downarrow \alpha \quad \alpha = \alpha_i}{\rho \vdash MN \Downarrow \beta_i}$$

down

$$\frac{\rho \vdash M \Downarrow down(<\alpha_1 \rightarrow V_1, \cdots, \alpha_k \rightarrow V_k>) \quad \rho \vdash N \Downarrow \alpha \quad \alpha = \alpha_i}{\rho \vdash MN \Downarrow V_i}$$

Table 2.-b Parallel Operational Semantics (continuing)

The rule for a μ-expression is simply to create a μ-closure. The rule for an application of μ-closure to an association sequence $<\alpha_1 \rightarrow V_1', \cdots, \alpha_k \rightarrow V_k'>$ first allocates a new communication variable β_i for each α_i, which corresponds to a node of the recursive datum to be created. Then it binds the variable f to a renaming function and d to a down function embedding the association sequence. The renaming function renames each old variable (α_i) to new one (β_i), and the down function receives a communication variable and returns the value associated with the variable, which corresponds to data transfer via inter-processor communication. Finally, the μ-expression is applied to each value V_i' to obtain a new value V_i simultaneously. The result of the μ-application is the

new recursive datum whose nodes are replaced by the new ones on which the corresponding newly computed values are placed, i.e. $<\beta_1 \rightarrow V_1, \cdots, \beta_k \rightarrow V_k>$.

This operational semantics is apparently unsafe — there are some terms that evaluate to *wrong*. However, we believe that the type system is sound with respect to this operational semantics, and therefore the operational semantics is safe for any type correct terms. We intend to present a more complete account for the calculus including type soundness and other formal properties elsewhere.

5 Implementation Strategies

We have proposed in Section 4.1 to represent distributed recursive data by global pointers and shown how the parallel constructs are executed when they applied to a distributed list. A naïve integration of the proposed method in an implementation of a functional language, however, does not necessarily respect the formal operational semantics, and some programs may yield incorrect result.

One way to overcome the difficulty is to restrict the calculus and introduce the notion of *virtual processor*. We describe this strategy in some detail below. The method works well for the restricted calculus, and execution obeys the formal operational semantics. We also discuss, assuming a specific network configuration, an implementation method for less restricted calculus.

5.1 A Restricted Calculus and Its Compilation

We impose the following restriction on the calculus.

- *Nested parallelism is prohibited.*
  μ-expression can not be applied in μ-expression's body.
- *Nested recursive data is not allowed.*
  Types of the form $\mu t.(\cdots \mu s.\cdots)$ are disallowed. This prohibits the use of the data types such as list of list.
- *Executing up and dn inside μ-expression is prohibited.*

These constraints considerably weaken the expressive power of the calculus. However, the restricted calculus is still worth considering, since it seems to preserve the enough power to express most of typical data parallel algorithms so far considered such as those described in Section 3.3. Furthermore, the restricted calculus can be implemented on a wide range of massively parallel machines without much difficulty.

Even with above restriction, there are some cases that a distributed recursive datum can not be expressed by global pointers, since more than two elements of the recursive datum may be allocated to a single processor. (Such a situation does not frequently arises. Indeed, the examples described so far does not create such a recursive datum.) To overcome this difficulty, we assume that computational power of each physical processor is shared by a non-empty set of *virtual processors* and that the number of virtual processors residing on a physical processor can be increased at run time. Computational power of a physical processor

is shared by letting each virtual processor execute the same set of instructions in a loop.

A program is now compiled as below. The program starts in the state that each physical processor's computational power is shared by a single virtual processor. A global pointer is now a pair of a virtual processor id and the pointer to the heap area. The execution method for *up* described in Section 4.1 has to be changed slightly as follows. To execute $up\ M\ with\ x_1 = N_1, \cdots, x_k = N_k\ end$, every virtual processor Q_1 first evaluates $(\lambda x_1 \cdots x_k.M)N_1 \cdots N_k$ to a value V_{Q_1}, and then selects the same virtual processor P, where no elements of recursive data have not been allocated to the physical processor on which processor Q_1 is residing. Suppose that, in each processor Q_1 other than the selected processor P, V_{Q_1} contains n global pointers $(P_1, p_1), \cdots, (P_n, p_n)$ as its immediate subterm and that each p_i is either q_1, q_2, or q_m where q_i's are distinct and $m \leq n$. If $m \geq 2$, processor Q_1 creates $m-1$ new virtual processors $Q_2, \cdots, Q_m$ sharing computational power of the physical processor on which the virtual processor Q_1 itself is residing. Then each processor $Q_i (1 \leq i \leq m)$ yields a global pointer (P, q_i). The execution method for *dn* also needs a slight extension. To execute $dn(M)$, all the second elements of the global pointers contained in the broadcasted datum as subterms of the recursive datum must be updated to the second element of the global pointer obtained by evaluating M. As for μ-application, the body of a μ-expression is compiled straight forwardly, since executing any parallel operations are not allowed by the restriction. However, there are some cases that an application of d is left unevaluated in a function closure and its evaluation is delayed. To correctly retrieve a remote datum when such a delayed application is evaluated, each global pointer (P, p) must be augmented with a local pointer, which points to the global pointer (Q, q) from which the global pointer (P, p) was obtained by applying a μ-expression.

Let us roughly estimate how efficiently the code compiled by the above method is executed. *up* can be executed in a constant time, since allocation of a new processor can be done by maintaining the same counter on every processor and increasing it. *dn* requires broadcasting a datum all over the processors, which seems to cause slow-down. For example, a hypercube connected multicomputer with p processors costs $O(\log p)$ time to execute a broadcast. However, broadcast by *dn* does not cause much slow down, since the calculus is designed to process recursive data by applying a parallel function to them iteratively, which eliminates a number of *dn* executed to unfold recursive data. In a typical program such as the suffix sum program, *dn* is used only once for each iteration applying a parallel function to a recursive datum, as a program written in a conventional array-based language is compiled to the code which globally synchronizes and computes global-or for each time executing a loop. *dn* therefore slows down execution at most by a constant factor compared to conventional array-based languages. μ-application can also be executed as efficiently as mapping a set of instructions over the elements of an array in a conventional data parallel language.

5.2 Toward Full Implementation

Implementing the full calculus constitutes a challenge. One of major difficulties is parallel resource allocation, which significantly complicates the data distribution strategy and consequently causes the machine slow down. Finding a general solution to this problem appears to be difficult. Here we offer a solution specific to hypercube connected multicomputers. This solution allows a restricted form of nested parallelism and nested distributed recursive data so that the calculus is still implemented reasonably efficiently while maintaining its power to express some parallel version of typical functions for nested recursive data such as map-functions.

Let d be the maximum nesting depth of the recursive types in a given program, which is determined by type checking. ($\mu t.unit + t \times (\mu s.unit + s)$ is, for example, a recursive type of depth 2.) Then the cube is divided into sub^{d-1}-cubes, where sub^0-cube is the whole cube, and sub^{n+1}-cube is obtained by dividing each sub^n-cube. The size of sub^i-cube for each i is determined so that the data may be distributed well. Nested recursive data are implemented in the following way. The scalar values are loaded on every processor; the recursive data of first depth are distributed over every processors so that each element of the data is loaded on every processor of a subcube; the recursive data of ith depth are, inductively, distributed over every processors of a sub^{i-1}-cube so that each element of the data is loaded on every processor of a sub^i-cube of the sub^{i-1}-cube. The nested parallelism is then allowed if the following condition is satisfied: each μ-expression of nesting level i (with regarding nesting level 0 as outermost μ-expression) is applied to the recursive datum distributed on a sub^i-cube.

6 Conclusion, Ongoing Work, and Further Investigation

We have proposed a typed functional calculus suitable for data parallel programming on massively parallel multicomputer systems. The calculus supports distributed recursive data and a parallel function application mechanism for those data. In this calculus, a new form of recursion, called *parallel recursion* is expressed, which overcomes inherent sequentiality of ordinary recursion, and exploits more parallelism in manipulating recursively defined data. We have developed a type system and parallel operational semantics for the calculus. We have described an SPMD execution model for the calculus, and proposed two implementation strategies based on it.

We believe that the calculus and its SPMD execution model provide a basis to design and implement a high level data parallel language. Such a claim needs to be substantiated by an actual implementation. We plan to develop a prototype language based on the proposed calculus. Our current strategy is to implement a restricted calculus by the method described in Section 5.1 by writing a translator that converts the restricted calculus to an existing data parallel language. We are designing a translator that converts ML like programming language extended

with data parallel constructs into TUPLE [Yua92], a data parallel extension of Lisp.

The calculus presented here offers a number of interesting challenges. One of them is to find an efficient way of creating distributed recursive data. The calculus has not overcome the difficulty of the inherent sequentiality of the recursive data creation, since *up* must be called sequentially. One approach is compile time distribution of recursive data using a syntax for direct representation of recursive data. A more interesting approach would be to develop a form of parallel I/O interface to an external world.

Acknowledgment

We would like to thank Kazuhiko Kato for insightful discussion about data parallel programming.

References

[Ble93] G.E. Blelloch. NESL: A nested data parallel language. Technical Report Tech. Report, CMU-CS-93-129, Carnegie Mellon University, 1993.

[For93] High Performance Fortran Forum. *High Performance Fortran language specification*, May 1993.

[Gup92] R. Gupta. SPMD execution of programs with pointer-based data structures on distributed-memory machines. *Journal of Parallel and Distributed Computing*, Vol. 16, pp. 92–107, 1992.

[Hal85] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, pp. 501–538, 1985.

[HF93] G. Hains and C. Foisy. The data-parallel categorical abstract machine. In *Proc. PARLE93: Parallel Architectures and Languages Europe (LNCS 694)*, 1993.

[HQ91] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, 1991.

[HS86] W.D. Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of ACM*, Vol. 29, No. 12, pp. 1170–1183, 1986.

[Kah87] G. Kahn. Natural semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science(LNCS 247)*, pp. 22–39. Springer Verlag, 1987.

[Kar87] A. Karp. Programming for parallelism. *IEEE Computer*, pp. 43–57, May 1987.

[Las86] C. Lasser. *The Essential *Lisp Manual*. Thinking Machine Corporation, Cambridge, MA, July 1986.

[RRH92] A. Rogers, J. Reppy, and L. Hendren. Supporting SPMD execution for dynamic data structurs. In *Proc. 5th International Workshop on Languages and Compilers for Parallel Computing (LNCS 757)*, 1992.

[RS87] J. Rose and Guy L. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL87-5, Thinking Machine Corporation, Cambridge, MA, 1987.

[Ryt85] W. Rytter. The complexity of twoway push down automata and recursive programs. In A. Apostolica and Z. Galil, editors, *Combinatorial Algorithms on Words*. Springer-Verlag, NATO ASI Series F:12 1985.

[Sab88] G. Sabot. *Paralation Lisp Manual*, May 1988.

[ST94] D. Suciu and T. Tannen. Efficient compilation of high-level data prrallel algorithm. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, June 1994.

[WS87] S. Wholey and Guy L. Steele Jr. Connection machine lisp: A dialect of common lisp for data parallel programming. In *Proc. International Conference on Supercomputing*, May 1987.

[Yua92] T. Yuasa. A SIMD environment TUPLE for parallel list processing. In *Parallel Symbolic Computing: Languages, Systems, and Applications (LNCS 748)*, pp. 268–286. Springer Verlag, 1992.

This article was processed using the LaTeX macro package with LLNCS style