J. Functional Programming **9** (4): 427–462, July 1999. Printed in the United Kingdom

*Parallel functional programming on recursively defined data via data-parallel recursion**

SUSUMU NISHIMURA † and ATSUSHI OHORI‡
Research Institute for Mathematical Sciences, Kyoto University,
Sakyo-ku, Kyoto 606-8502, Japan
(*e-mail:* {nisimura,ohori}@kurims.kyoto-u.ac.jp)

Abstract

This article proposes a new language mechanism for data-parallel processing of dynamically allocated recursively defined data. Different from the conventional array-based data-parallelism, it allows parallel processing of general recursively defined data such as lists or trees in a functional way. This is achieved by representing a recursively defined datum as a system of equations, and defining new language constructs for parallel transformation of a system of equations. By integrating them with a higher-order functional language, we obtain a functional programming language suitable for describing data-parallel algorithms on recursively defined data in a declarative way. The language has an ML style polymorphic type system and a type sound operational semantics that uniformly integrates the parallel evaluation mechanism with the semantics of a typed functional language. We also show the intended parallel execution model behind the formal semantics, assuming an idealized distributed memory multicomputer.

Capsule Review

Data parallelism is a simple and effective method for programming massively parallel computer systems: the data are distributed across the processors, allowing the program to perform operations on all the data elements simultaneously. Traditional data parallel languages require the data elements to be organised in a flat structure, such as a vector or matrix. This paper offers a method for generalising data parallelism to recursively defined data structures such as trees. The essential idea is a generalisation to user-defined recursive data structures of the pointer-jumping method for traversing a linked list in log time on a parallel machine.

* A preliminary summary of some of the results of this article appeared in the *Proceedings of the Workshop on Theory and Practice of Parallel Programming*, Sendai Japan (*LNCS 907*, 1995, under the title "A Calculus for exploiting data parallelism on recursively defined data."

† Susumu Nishimura's work was partly supported by the Japanese Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Encouragement of Young Scientists, 10780187, 1998.
‡ Atsushi Ohori's work was partly supported by the Japanese Ministry of Education Grant-in-Aid for Scientific Research on Priority Area no. 275 "Advanced databases", and by the Parallel and Distributed Processing Research Consortium, Japan.

1 Introduction

Data-parallelism is widely recognized as one of most practical parallel programming paradigm, and a number of data-parallel programming languages have been developed and used in practice. The central idea underlying this paradigm is to evaluate a uniform collection of data, typically an array, by simultaneously applying the same function to each data element in the collection. The importance of this paradigm is twofold. First, data-parallel programs are easy to understand. In data-parallel languages, parallelism is coupled with primitive operations for some collection data types. By this coupling, programmers can have a high level view of single threaded execution of data-parallel programs: a data-parallel operation can be viewed as a single operation for a collection of data. Secondly, in contrast to the programmer's viewpoint of single threaded execution, there is also a simple and effective parallel execution model for data-parallel programs, which is called SPMD execution model (Karp, 1987). In the SPMD model, data elements in a collection are scattered over the processors in advance, and every processor in a parallel machine executes the same program on those scattered data elements. The data-parallel primitive operations are executed instantaneously by simultaneously operating on the portion of data elements assigned to each processor.

One limitation of most of currently available data-parallel languages is that the parallelism is restricted to some particular built-in data structures such as arrays or sequences. Conventional data-parallel languages have been developed from an existing imperative language by embedding a set of parallel evaluation primitives for arrays. Examples include C* (Rose and Steele Jr., 1987), Dataparallel C (Hatcher and Quinn, 1991), and High Performance Fortran (Forum, 1993). There are also a number of proposals for functional data-parallel languages such as Connection Machine Lisp (Wholey and Steele Jr., 1987), *Lisp (Lasser, 1986), Paralation Lisp (Sabot, 1988), TUPLE (Yuasa, 1992), Plural EuLisp (Merrall and Padget, 1992), DPML (Hains and Foisy, 1993), Nesl (Blelloch, 1993), FX (Talpin and Jouvelot, 1993), and Caml Flight (Foisy and Chailloux, 1995). These languages enable us to exploit data-parallelism in functional programming, but the source of data-parallelism in those languages is still limited to predefined built-in collection data types.

The approach of restricting data-parallelism to a particular built-in data type has the obvious advantage of ease of implementation and optimization. As demonstrated by functional data-parallel languages mentioned above, it can also achieve a reasonable integration of functional programming and data-parallelism relatively easily. Unfortunately, however, this limited approach to parallel programming cannot take full advantage of the benefits of functional programming. One of important features of modern functional programming languages is the ability to define new data types using recursive data type declaration and to use them with recursive functions. In most of typed higher-order languages such as SML (Milner *et al.*, 1990) and Haskell (Hudak *et al.*, 1992), this feature is so pervasive that it is almost impossible to imagine a large practical program without using this feature. Data-parallelism should ideally be integrated in a functional programming language so that the programmer can freely define appropriate data structures for the application and write functional data-parallel programs operating on them. The goal of this paper

is to achieve such integration by developing programming language constructs for data-parallel evaluation of general recursively defined data, and integrating them in a higher-order functional programming language with recursive types.

A crucial problem in achieving this goal is to find a proper mechanism for applying the same operation to all the data elements in a given general recursive data structure. For a particular data structure, it is not hard to design a set of parallel primitives for this purpose. For simple linear data structures such as arrays or lists, for example, it would be enough to provide some commonly used data-parallel operations like map, scan, reduction, etc. For general recursively defined data, however, finding a suitable mechanism for parallel evaluation constitutes a non-trivial technical challenge.

In a conventional sequential machine, recursive data structures are implemented as linked data structure and are processed by recursive functions. However, this commonly adopted framework of recursive data processing becomes an obstacle for effective parallelization. Conventionally, recursive functions can be parallelized by annotating each recursive call with the speculative evaluation primitives, most notably `future` and `touch` in Multilisp (Halstead, 1985). After the proposal of Multilisp, various refinements and implementation methods have been proposed. For example, PaiLisp (Ito and Matsu, 1988) provides more general speculative evaluation primitives, and Olden (Rogers *et al.*, 1995) allows speculative evaluation in a distributed environment using process migration technique. Unfortunately, the speculative evaluation strategy does not scale up to a large size of data. To see the problem, consider the following recursive function (written in ML-like syntax), which sums up all the elements in a given integer list:

```
fun sum L = case L of Nil => 0
                    | Cons(n,tl) => n+(sum tl)
            end;
```

Even if all the recursive function applications are speculatively evaluated, this function still requires the time proportional to the length of the given list due to the inherent sequentiality of a recursive function operating on recursively defined data: a recursive function forces a series of nested recursive calls to be processed sequentially. To extend data-parallelism to general recursively defined data, we need an alternative mechanism for parallel evaluation.

In this paper, we develop such a mechanism based on the following observation. Instead of computing the desired value directly from the given recursively defined value, we can equivalently compute a *system of equations* that defines the desired result from a system of equations that defines the given datum by repeatedly transforming systems of defining equations. To exploit this idea, we represent a recursively defined datum as a system of equation of the form

$$X_i.\langle X_1 = C_1(\ldots), \ldots, X_i = C_i(\ldots), \ldots, X_k = C_k(\ldots)\rangle$$

where each $X_j (1 \leqslant j \leqslant k)$ is a variable indexing a node of the recursive datum whose constructor is $C_j(\ldots)$ possibly containing references to other nodes through $X_1, \ldots, X_k$, and the distinguished variable X_i indexes the root node. This represen-

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

tation can be regarded as a generalization of Gupta's (1992) representation of a dynamically allocated linked data structure in a distributed environment using global names. In a *sequential* functional language, this representation may only introduce extra overhead by eliminating the possibility of sharing substructures. It also makes it difficult to incrementally construct new datum from existing one. However, this opens up a new possibility of parallelism, as seen in the following.

(i) By giving a concrete representation to a system of equations, we can treat them as a uniform collection of data and therefore can transform it in a data-parallel fashion.
(ii) Since any recursive datum can be represented as a system of equations and the transformation operation on equations can be represented as a function, this transformation mechanism serves as a general way to express data-parallel computation on arbitrary recursive data in a functional style.
(iii) By simply regarding a system of equations as an array of equations, we obtain an SPMD execution model for general recursively defined data.

Based on this observation, we develop a general language mechanism for data-parallel evaluation of recursively defined data and integrate them in a higher-order typed language. As we explain in detail later, the mechanism we propose can be regarded as a generalization of the so called pointer jumping technique (Jájá, 1992), which has been employed by many data-parallel algorithms on linked data structure. So far the existing presentations of this technique are all based on imperative manipulation of pointer arrays. Our recursive data representation and data-parallel transformation mechanism provide a way to express and understand them in a more declarative style.

We carry out our technical development in a formal framework of the typed lambda calculus with recursive types. The proposed language has a rigorous semantics that accounts for parallel evaluation of recursively defined data. There are some attempts to give a formal semantics for a language with data-parallel constructs. Hains *et al.* (1993) have given an execution model of data parallel categorical abstract machine (DPCAM), whose prototype implementation is Caml Flight (Foisy and Chailloux, 1995). The DPCAM intends the SPMD execution of an array of sequential CAMs communicating with each other through a special primitive `get`. Later, they proposed a few variants of more formal semantics for abstract computation models based on the DPCAM (Loulergue and Hains, 1997; Loulergue *et al.*, 1998). Hammarlund and Lisper (1993) have given a mathematical definition of some data-parallel primitives that operate on a set of indices in parallel. Suciu and Tannen (1994) have provided a clean formal semantics for their data-parallel language which contains sequences and simple parallel primitives to map functions over sequences. In contrast to those approaches, which presume a specific data structure for data-parallelism such as arrays, our language allows arbitrary recursive data to be processed by the general data-parallel constructs and is therefore more general.

We should emphasize that our aim in this paper is not to invent new parallel algorithms or optimization methods for various recursively defined data, but to provide a general language construct for expressing data-parallel operations on general

recursive data in a uniform and declarative way. To write data-parallel programs on a particular recursively defined data, one must of course invent some algorithms that efficiently work on the data structure, which is a task heavily depending on algebraic properties of individual data structures. Our contribution should therefore be complemented by the researches on developing new data structures, algorithms and optimization methods for efficient data-parallel execution. Toward this direction, there are several relevant recent approaches for functional data-parallel programming. O'Donnell (1993) has shown that a flexible array structure, extensible sparse functional arrays, can be efficiently implemented on massively parallel computers. Misra (1994) has shown that many data-parallel algorithms can be expressed as a recursion of data-parallel operations on a specific data structure called powerlist, which is a list constructed by concatenating two lists of the same length. Darlington *et al.* (1993) have proposed using functional skeletons for optimizing parallel functional programs by program transformation. Skeletons, which have been introduced by Cole (1989), are a set of high level templates for typical parallel programs, and functional skeletons provides skeletons as higher-order functions whose behavior is parameterized by function arguments. There are also algebraic approaches for optimizing parallel programs (Bratvold, 1994; Skillicorn, 1994) and for parallelizing sequential programs (Hu *et al.*, 1998) based on the general framework of Bird–Meertens formalism (Bird, 1987) for program transformation. We believe that our language could serve as a framework to represent some of these results. For example, polymorphic data-parallel functions can be used to define some functional skeletons, and the program transformation technique could be applied to our language constructs.

The rest of the paper is organized as follows. In section 2, we explain the basic idea of our data-parallel evaluation mechanism for recursively defined data. Section 3 defines the language, and informally explains the intended behavior of the language constructs. In section 4, we formally give a type system and an operational semantics for the language and then show that the type system is sound with respect to the operational semantics. Section 5 demonstrates how some typical data-parallel algorithms are expressed in our language. Section 6 shows a simple SPMD execution model for our language. Finally, section 7 concludes the paper.

2 Data-parallel recursion for recursively defined data

To analyze the desirable properties of data-parallel functions for recursively defined data, let us consider again the function `sum` defined in the previous section. The semantics of the recursive definition of `sum` implies that the application of `sum` to an n element list yields a sequence of n recursive calls of `sum`, and therefore straightforward parallelization of the recursive calls still requires a series of n communications that must be processed sequentially. However, there is a data-parallel algorithm, originally due to Wyllie (1979), which computes the sum of an n element integer list in $O(\log n)$ parallel steps. Here we use the presentation by Hillis and Steele Jr. (1986). The main idea behind 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 pseudo code describes such an algorithm

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

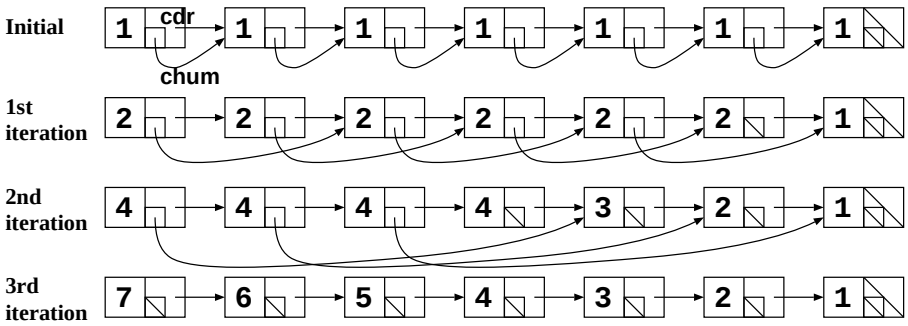

Fig. 1. Data-parallel suffix sum of an integer list.

that computes the suffix sum of an integer list with extra `chum` pointer, where `value` and `next` pointers 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
```

Figure 1 illustrates how the suffix sum of a list is computed by this algorithm. In each iteration, the partially computed value (`value[k]`) is combined with another partially computed value (`value[chum[k]]`), and then all the `chum` pointers are doubled. Iteration is repeated until all the `chum` pointers are set to `nil`. This technique is often called *pointer jumping* and is widely used for data-parallel algorithms on linked data structures.

The new recursive construct proposed in this paper is based on the observation that the pointer jumping technique indicates a 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 a list consisting of elements of type α is represented by the recursive type $\mu t.unit + \alpha \times t$ where *unit* is the type of the empty list, and $+$ and $\times$ are disjoint union and product constructor, respectively. A recursive function f of type $(\mu t.unit + \alpha \times t) \to \tau$ in general has the following structure:

$$fix\ f.\lambda x.case\ x\ of\ inl(*) \Rightarrow c,\ inr((h,t)) \Rightarrow F\ h\ (f\ t)$$

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

$$f\ L_1 = F\ a_1\ (f\ L_2),\ f\ L_2 = F\ a_2\ (f\ L_3),\ \cdots,\ f\ L_n = F\ a_n\ (f\ L_{n+1}),\ f\ L_{n+1} = c$$

The data-parallel suffix sum algorithm can be understood as parallelization of such

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

$$\chi_1 \xrightarrow{\text{parallel map}} \chi_2 \xrightarrow{\text{parallel map}} \cdots \xrightarrow{\text{parallel map}} \chi_n$$

data-parallel recursion

(χ_j: a system of equations representing a recursive datum)

$$\chi : \left\{ \begin{array}{ccc} X_1 = C_1(\ldots) & \xrightarrow{f} & C'_1(\ldots) = X'_1 \\ \vdots & \vdots & \vdots \\ X_n = C_n(\ldots) & \xrightarrow{f} & C'_n(\ldots) = X'_n \end{array} \right\} : \chi'$$

parallel map

Fig. 2. Data-parallel recursion.

a sequence of recursive function calls, as shown in the following. By indexing each application $f\ L_i$ by X_i as an unknown value, the above sequence of function calls can be considered as a system of equations over the set of unknowns $X_1, \ldots, X_{n+1}$, as below:

$$X_1 = F\ a_1\ (X_2),\quad X_2 = F\ a_2\ (X_3),\ \cdots, X_n = F\ a_n\ (X_{n+1}),\quad X_{n+1} = c$$

Let G_i be the partially applied function $F\ a_i$, and for any $i \leqslant j$ let $G_{i,j}$ be the composition of functions $G_i \circ G_{i+1} \circ \cdots \circ G_{j-1} \circ G_j$. Then, we can compute the result $f\ L = X_1 = G_{1,n}(c)$ by repeatedly transforming the system of equations in the following way:

$$\begin{array}{rllll} (\textit{initial}) & X_1 = G_{1,1}(X_2) & X_2 = G_{2,2}(X_3) & \cdots & X_n = G_{n,n}(X_{n+1}) \quad X_{n+1} = c \\ (1\textit{st}) & X_1 = G_{1,2}(X_3) & X_2 = G_{2,3}(X_4) & \cdots & X_n = G_{n,n}(c) \quad X_{n+1} = c \\ (2\textit{nd}) & X_1 = G_{1,4}(X_5) & X_2 = G_{2,5}(X_6) & \cdots & X_n = G_{n,n}(c) \quad X_{n+1} = c \\ \vdots & \vdots & \vdots & & \vdots \quad \vdots \\ (\log n\,\textit{th}) & X_1 = G_{1,n}(c) & X_2 = G_{2,n}(c) & \cdots & X_n = G_{n,n}(c) \quad X_{n+1} = c \end{array}$$

Transformation of a system of equations is done by applying the same function to each equation simultaneously. In the case that the composition $G_{i,j} \circ G_{j+1,k}$ is computed in constant time resulting in a function $G_{i,k}$ whose application is computed in constant time, the above method computes the application $f\ L$ in $O(\log n)$ parallel steps. The special pointers `chum` used in the data-parallel suffix sum algorithm can be regarded as a representation of unknowns X_i.

The above observation leads us to a new notion of recursion for data-parallelism, which we call *data-parallel recursion*. As illustrated in Figure 2, data-parallel recursion is a series of transformations on a system of equations (denoted by χ, χ', etc. in the figure) representing a recursively defined datum. Each transformation is done by a data-parallel operation called *parallel map*, which applies the same function f simultaneously to each equation in a given system of equations to yield another system of equations. Since every recursive datum is uniformly expressed by a system of equations, data-parallel recursion can be applied to arbitrary recursively defined

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

$M ::=$	c	(constant)
	x	(identifier)
	$fun\ f(x) \Rightarrow M$	(recursive function)
	MM	(function application)
	(M, M)	(pair)
	$fst(M)$	(first component)
	$snd(M)$	(second component)
	$inl(M)$	(left injection)
	$inr(M)$	(right injection)
	$case\ M\ of\ inl(x) \Rightarrow M, inr(y) \Rightarrow M$	(case branch)
	$let\ x = M\ in\ M$	(polymorphic let)
	$foreach\ x \in M\ with\ [f, d]\ do\ M$	(parallel map)
	$up\ M\ with\ x_1 = M_1, \ldots, x_m = M_m\ end$	(recursive data constructor)
	$dn(M)$	(recursive data destructor)

Fig. 3. Syntax of the language.

data types. Furthermore, it has the conceptual generality that will serve as a basic tool for describing data-parallel algorithms for recursively defined data.

To integrate data-parallel recursion in the framework of 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 we have explained, and we provide a language construct for defining data-parallel operation on those recursively defined data. The basic idea to define a data-parallel operation is to specify a function that transforms each datum in a given system of equations ($C_i(\ldots)$ in Figure 2) into another datum ($C'_i(\ldots)$). This language construct is introduced as a term constructor that lifts a constructor transformer function of type $\tau(r) \to \tau'(s)$ to a function of type $\mu t.\tau(t) \to \mu t.\tau'(t)$. In the following sections, we will formally define the language where these constructs are available and give an operational semantics that accounts for the data-parallel evaluation of recursive types.

3 The data-parallel functional language

The syntax of our data-parallel language is given in Figure 3. It is a variant of the lambda calculus with constants, products and sums. We assume constants include $*$ of type *unit*. Lambda abstraction is combined with recursion in $fun\ f(x) \Rightarrow M$, which should be understood as a function $\lambda x.M$ where f is bound to the function itself in M. The language is intended to be typed implicitly, and the let expression is for introducing ML-style polymorphism in the underlying type system.

The last three constructs are for our data-parallel recursive data manipulation. In the rest of this section, we explain our recursive data representation and then describe the behavior of each data-parallel construct in turn. For each of them, we also describe the necessary typing constraint and briefly give the idea of parallel execution. A formal type system and the operational semantics of the language will be defined in section 4, and more detailed description of parallel execution will be given in section 6.

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

3.1 Representation of recursive data

As explained in section 2, every recursive datum is represented as a system of equations in our language. Each system of equations is expressed by a run-time value of the following form.

$$X.\langle X_1 \to V_1, \ldots, X_k \to V_k \rangle \qquad (k \geqslant 1,\ X = X_i \text{ for some } i)$$

This represents a system of k equations over variables $X_1, \ldots, X_k$, and each $X_i \to V_i$ enclosed in the angle brackets is an equation, which reads "the variable X_i is associated to a value V_i." We call $X_1, \ldots, X_k$ *node variables* and $V_1, \ldots, V_k$ *node values*. We call this representation *association sequence*. An association sequence is always prefixed by a node variable indicating the root node of the recursive datum. The type for the sequence is roughly given as follows: $X.\langle X_1 \to V_1, \ldots, X_k \to V_k \rangle$ has type $\mu t.\tau(t)$ if V_i has type $\tau(t)$ for all i under the assumption that the node variables $X_1, \ldots, X_k$ have type t. For example, the following is a list $[1, 2, 3]$ represented by an association sequence of type $\mu t.unit + int \times t$.

$$X_1.\langle X_1 \to inr((1, X_2)), X_2 \to inr((2, X_3)), X_3 \to inr((3, X_4)), X_4 \to inl(*) \rangle.$$

The rationale of this representation is the following. The association sequence $X.\langle X_1 \to V_1, \ldots, X_k \to V_k \rangle$ models a data distribution over the processors in such a way that each data element V_i is allocated to a distinct processor identified by X_i. The data distribution enables parallel map to be applied in constant time, and any data element V_i can be accessed from any processor by communicating with the processor identified by X_i.

3.2 Constructs for recursive data manipulation

To support data-parallel manipulation of recursive data represented as above, the language provides the constructs for transforming, constructing, and destructing those data.

3.2.1 Parallel map: Data-parallel recursive data transformation

The most important data-parallel construct is what we call parallel map.

foreach $x \in N$ *with* $[f, d]$ *do* M

This transforms a recursive data N represented by a system of equations to another recursive data of possibly different recursive type in a data-parallel fashion. The data-parallel transformation is achieved by simultaneously applying the same function $\lambda x.M$ to each node value in the given system of equations. (This is equivalent to creating a binding of x to each node value and evaluating M simultaneously under each binding.) During the parallel function application, the identifiers f and d are used as functions for special purposes explained below.

The parallel map construct first evaluates N to an association sequence $X_i.\langle X_1 \to V_1, \ldots, X_k \to V_k \rangle$ of type $\mu t.\tau(t)$. Before data-parallel execution, the node names

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

$X_1, \ldots, X_k$ in each node value V_j are replaced by temporary node names $A_1^k, \ldots, A_k^k$, respectively. Temporary node names act like atomic values. The replacement procedure is necessary for obtaining a type sound semantics for the language. (We later give a formal definition of the temporary node names in section 4.2.) To assign types to those names, we introduce special type variable, called *node type variable* (ranged over by n). The set of newly introduced temporary names are typed by a fresh node type variable n. Let $V_1', \ldots, V_k'$ be the values obtained by the replacement. Each V_j' has type $\tau(n)$. The map function $\lambda x.M$ is then simultaneously applied to each V_j' to produce a system of equations constituted by a set of new node variables $Y_1, \ldots, Y_k$, which are assigned to a fresh recursive type variable s. During the evaluation of the function applications, the system binds variable f to the *renaming function* of type $n \to s$ which maps each temporary node name A_j^k to the corresponding new node variable Y_j, and the system binds variable d to the *down function* of type $n \to \tau(n)$ which maps each temporary node name A_j^k to the corresponding value V_j'. The final result of the parallel map is an association sequence $Y_i.\langle Y_1 \to V_1'', \ldots, Y_k \to V_k'' \rangle$, where each V_j'' is the result of applying the function to V_j'. All V_j'''s have the same type $\tau'(s)$, and therefore the type of the resulting recursive datum is $\mu s.\tau'(s)$.

Assuming the distributed data representation of the system of equations, we can trigger the data-parallel computation by simultaneously applying the same function $\lambda x.M$ to the data element allocated to every processor. The distinguished feature of the parallel map is the special functions, especially the down function. The down function can be used to retrieve the data element allocated to the processor identified by a temporary node name A_j^k.

A function that simply maps a given function over the elements of a given list is written via parallel map as follows.

$$\begin{array}{l}\lambda F.\lambda L.foreach\ x \in L\ with\ [f,d]\ do \\ \qquad case\ x\ of\ inl(y) \Rightarrow inl(*), \\ \qquad\qquad inr(y) \Rightarrow inr((F\,fst(y), f\,snd(y)))\end{array}$$

This map function applies the function F simultaneously to each value in the list L. The renaming function f is applied to temporary names contained in the cdr part of each cell to retain the *links* in the resulting list. The following is an example which uses the down function d:

$$\begin{array}{l}foreach\ x \in L\ with\ [f,d]\ do \\ \quad case\ x\ of\ inl(y) \Rightarrow inl(*), \\ \qquad inr(y) \Rightarrow let\ p = case\ (d\,snd(y))\ of\ inl(z) \Rightarrow false, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad inr(z) \Rightarrow (fst(y) = fst(z)) \\ \qquad\qquad\qquad in\ inr((p, f\,snd(y)))\end{array}$$

This parallel map transforms an integer list L to a boolean list. It tests whether the integer in each cell is equal to the integer in the next cell, and returns the result of the tests as a boolean list. Suppose L is an integer list $[1,2,2]$, represented as

$$X_1.\langle X_1 \to inr((1, X_2)), X_2 \to inr((2, X_3)), X_3 \to inr((2, X_4)), X_4 \to inl(*) \rangle.$$

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

The parallel map applies the same function simultaneously to each of the values $inr((1, A_2^4))$, $inr((2, A_3^4))$, $inr((2, A_4^4))$, $inl(*)$. To obtain the next cell, the down function d is applied to a temporary node name to get the value associated to the temporary name. Finally, each temporary name A_j^4 is coerced to the corresponding new node variable Y_j by applying the renaming function f. The final result of the parallel map is the new system of equations:

$$Y_1.\langle Y_1 \to inr((false, Y_2)), Y_2 \to inr((true, Y_3)), Y_3 \to inr((false, Y_4)), Y_4 \to inl(*)\rangle,$$

i.e. a boolean list $[false, true, false]$.

Note that parallel map allows the user to write a parallel function that transforms recursively defined data by simply writing a function that maps each node value to another node value. The applicability of this mechanism is not limited to any particular presupposed data types such as lists or sequences; we can write parallel function that transforms arbitrary recursive datum of type $\mu t.\tau(t)$ to another recursive datum of type $\mu t.\tau'(t)$ by writing a function of type $\tau(n) \to \tau'(s)$. This enables us to write data-parallel programs for arbitrary recursive data in a uniform way by means of data-parallel recursion explained in section 2.

3.2.2 Recursive data constructor and destructor

To understand our recursive data constructor and destructor, one should note the difference between our recursive data representation and the usual one. Recursive types are usually understood as types identified up to folding and unfolding, i.e. as types satisfying the isomorphism $\mu t.\tau(t) \cong \tau(\mu t.\tau(t))$. In this view, to process recursive data with ordinary recursion, it is sufficient to simply introduce recursive data constructor and destructor as type coercion operators *Up* of type $\tau(\mu t.\tau(t)) \to \mu t.\tau(t)$ and *Down* of type $\mu t.\tau(t) \to \tau(\mu t.\tau(t))$. That means, a recursive datum of type $\mu t.\tau(t)$ and its unfolded type $\tau(\mu t.\tau(t))$ have the same data representation. In contrast to this, the data representations corresponding to the two types are completely different in our language: The former is an association sequence whose data elements are scattered over the processors, but the latter is a scalar value. Therefore our recursive data constructor and destructor are not simple type coercion operators but they are operators for conversion between the two data representations.

The counterpart of the conventional recursive data construction is expressed in our language as follows.

$$up\ M\ with\ x_1 = N_1, \ldots, x_m = N_m\ end$$

This expression is intended to create a new recursive datum of type $\mu t.\tau(t)$ by combining the m recursive data $N_1, \ldots, N_m$ as sub-data. The apparent difference from the conventional one is that it explicitly names the sub-data $N_1, \ldots, N_m$ by the identifiers $x_1, \ldots, x_m$ through which the sub-data are referenced in M. The explicit naming is needed because of our specific representation of recursive data: each recursive datum is represented as a system of equations where a set of node variables is used to identify the nodes of the recursive datum. Hence, to create a

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

recursive datum with a new root node, the system need to know the node variables by which sub-data are referenced.

To evaluate the expression, each sub-data N_i is evaluated to yield an association sequence $X^i.\langle X^i_1 \to V^i_1, \ldots, X^i_m \to V^i_m\rangle$. The identifiers $x_1, \ldots, x_m$ are bound to $X^1, \ldots, X^m$, respectively, and then M is evaluated to a value V. The value V will usually contain node variables $X^1, \ldots, X^m$ to refer to the root node of the corresponding recursive data. Finally, the new recursive datum is created as an association sequence which is a merged sequence of the m sequences of the sub-data with an additional association $X \to V$, where X is a new node variable designating the root node (i.e. X is prefixed on the sequence.) The expression should satisfy the following typing constraint: for the expression to have a type $\mu t.\tau(t)$, $N_1, \ldots, N_m$ should have the same type $\mu t.\tau(t)$ and M should have type $\tau(t)$ under the assumption that the identifiers $x_1, \ldots, x_m$ have type t.

Assuming every data element is allocated to a distinct processor, the idea of parallel execution of the recursive data constructor is rather simple. First, let all the processor select the same single processor to which any data element has not been allocated. Then the result of evaluation of M is allocated to the processor, which constitutes the association for the new root node. The association sequences are automatically merged, as all the processors resume execution of the subsequent computation. (See section 6.3 for more details.)

Here one should note that the construction of recursive data in our language still remains sequential: for example, consider the construction of a two-element list $L = [1, 2]$. (We simply write $up(M)$ instead of $up\ M\ with\ \ end$ in the rest of the paper.)

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

This yields the following result:

$$X_1.\langle X_1 \to inr((1, X_2)), X_2 \to inr((2, X_3)), X_3 \to inl(*)\rangle$$

As this example shows, to construct a list of length n from scratch, we need to apply the recursive data constructor $n + 1$ times, which is executed sequentially. However, our language does not intend to build a recursive datum from scratch. Instead, it assumes the recursive data to be processed are given in advance. For example, a recursive datum can be given in the program text statically by using some conventional syntax like $[1, 2, 3, 4, 5]$ for list, which enables the construction of recursive datum at compile time. For run-time construction of recursive data, one should be able to introduce, for example, a primitive operation that generates a list of a given number of a fixed element in a constant time.

The following expression is the recursive data destructor of our language:

$dn(M)$

This expression not only coerces type from $\mu t.\tau(t)$ to type $\tau(\mu t.\tau(t))$, but also changes the representation of the recursive datum M to its unfolded form as follows. Suppose M is evaluated to an association sequence $X_i.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle$. Then the recursive data destructor returns a value V'_i obtained from V_i by substituting the

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

association sequence $X_j.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle$ for each node variable X_j in V_i. For example, $dn(L)$, where L is the list defined above, is evaluated to the value: $inr((1, X_2.\langle X_1 \to inr((1, X_2)), X_2 \to inr((2, X_3)), X_3 \to inl(*)\rangle))$. The parallel execution of the unfolding by substitution involves broadcasting, since the unfolding is a conversion from a collection of data, i.e. an association sequence, to a scalar value which should be copied to all the processors. (See section 6.3 for details.)

4 Type system and operational semantics for the language

We present an ML-style polymorphic type system and an operational semantics of the language. We then show that the type system is sound with respect to the operational semantics.

In what follows, we use the following notations on sets and functions. The set difference is written $A \setminus B$. For a function f, its domain and the range (codomain) are written as $Dom(f)$ and $Range(f)$ respectively. A finite function f such that $Dom(f) = \{x_1, \ldots, x_n\}$ and $f(x_i) = v_i$ is written as $\{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\}$. For any set A such that $A \subseteq Dom(f)$, we define $f(A) = \{f(x) \mid x \in A\}$. For any functions f, g, we define $f + g$ as the following function: $Dom(f + g) = Dom(f) \cup Dom(g)$ and for all $x \in Dom(f + g)$, $(f + g)(x) = g(x)$ if $x \in Dom(g)$; otherwise $(f + g)(x) = f(x)$.

4.1 Type system

We assume that the meta-variable b ranges over a given set of base types including the type *unit*. The type system contains three kinds of type variables: *polymorphic type variables* (ranged over by α) to represent ML-style polymorphism, *recursive type variables* (ranged over by t) to represent recursive types of the form $\mu t.\tau$, and *node type variables* (ranged over by n) to represent temporary names used in the parallel map construct.

Following Damas and Milner (1982), the set of types is divided into the set of *monotypes* (ranged over by τ) and the set of *polytypes* (ranged over by σ) given by the syntax.

$$
\begin{array}{lll}
\tau ::= & b & \text{(base types)} \\
\mid & \alpha & \text{(polymorphic type variable)} \\
\mid & t & \text{(recursive type variable)} \\
\mid & n & \text{(node type variable)} \\
\mid & unit & \text{(unit)} \\
\mid & \tau \to \tau & \text{(function type)} \\
\mid & \tau \times \tau & \text{(product type)} \\
\mid & \tau + \tau & \text{(sum type)} \\
\mid & \mu t.\tau & \text{(recursive type)}
\end{array}
$$

$$
\begin{array}{lll}
\sigma ::= & \tau & \\
\mid & \forall \alpha.\sigma &
\end{array}
$$

The constructs $\forall \alpha.\sigma$ and $\mu t.\tau$ binds polymorphic type variable α in σ and recursive type variable t in τ respectively. The set of *free type variables* in a type τ, denoted

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

Table 1. *Typing rules*

constant

$$\frac{\text{if constant } c \text{ has type } \tau}{\Gamma \vdash c : \tau}$$

identifier

$$\frac{x \in Dom(\Gamma) \quad \tau \preceq \Gamma(x)}{\Gamma \vdash x : \tau}$$

function

$$\frac{\Gamma + \{f : \tau_1 \to \tau_2, x : \tau_1\} \vdash M : \tau_2}{\Gamma \vdash fun\ f(x) \Rightarrow M : \tau_1 \to \tau_2}$$

application

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

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

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

let

$$\frac{\Gamma \vdash M : \tau' \quad \Gamma + \{x : Clos_\Gamma \tau'\} \vdash N : \tau}{\Gamma \vdash let\ x = M\ in\ N : \tau}$$

up

$$\frac{\begin{array}{c}\Gamma + \{x_1 : s, \cdots, x_k : s\} \vdash M : \tau(s) \\ \Gamma \vdash N_i : \mu t.\tau(t) \ \text{ for each } i,\ 1 \leqslant i \leqslant m\end{array}}{\Gamma \vdash up\ M\ with\ x_1 = N_1, \ldots, x_m = N_m\ end : \mu t.\tau(t)}$$

(s is a new recursive type variable.)

dn

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

foreach

$$\frac{\begin{array}{c}\Gamma \vdash M : \mu t.\tau(t) \\ \Gamma + \{f : n \to s, d : n \to \tau(n), x : \tau(n)\} \vdash M : \tau'(s) \\ n \text{ does not occur free in } \tau'(s)\end{array}}{\Gamma \vdash foreach\ x \in M\ with\ [f, d]\ do\ N : \mu t.\tau'(t)}$$

(s and n are new recursive type variable and node type variable, respectively.)

by $FTV(\tau)$, is the set of polymorphic type variables not bound by any $\forall$. We identify types that differ only in the names of bound type variables and further adopt the usual "bound variable convention" on (both polymorphic and recursive) type variables, i.e. we assume that the set of all bound type variables are distinct and are different from any free type variables and that this property is preserved by substitution. A type substitution θ for free polymorphic type variables is a

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

function from a finite set of polymorphic type variables to monotypes. We write $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_k \mapsto \tau_k]$ for the substitution that maps each α_i to τ_i. This substitution for polymorphic type variables is extended to the substitution for monotypes by the obvious induction on the structure of monotypes. The result of applying a substitution θ to a polytype $\forall\alpha.\sigma$ is the type obtained by applying θ to its all *free type variables*. Under the bound type variable convention, we can simply take $\theta(\forall\alpha.\sigma) = \forall\alpha.\theta(\sigma)$. The analogous definition apply to type substitutions for recursive type variables. We write $\tau(t)$ for a type τ which possibly has free occurrences of t, and write $\tau(\tau')$ for the type obtained by substituting τ' for every free occurrences of t in τ.

The type system for the language is defined by a set of rules in Table 1 to derive a typing of the form

$$\Gamma \vdash M : \tau$$

where Γ is a type environment, which is a finite function from identifiers to polytypes. We define $FTV(\Gamma)$ to be the set of free type variables that appear in Γ, i.e. $FTV(\Gamma) = \bigcup_{x \in Dom(\Gamma)} FTV(\Gamma(x))$. The relation $\sigma \geq \tau$ indicates that τ is an *instance* of σ, i.e. $\sigma = \forall\alpha_1 \ldots \alpha_m.\tau'$ for some τ' and there exists a type substitution θ such that $Dom(\theta) = \{\alpha_1, \ldots, \alpha_m\}$ and $\tau = \theta(\tau')$. $Clos_\Gamma \tau$ is *generalization* of τ w.r.t. type environment Γ, i.e. $Clos_\Gamma \tau = \forall\alpha_1 \ldots \alpha_m.\tau$ where $\{\alpha_1, \ldots, \alpha_m\} = FTV(\tau) \setminus FTV(\Gamma)$.

4.2 Operational semantics

We present a formal operational semantics for the language in natural semantics (Kahn, 1987).

4.2.1 Semantic values and environments

To represent a system of equations, we assume that there is a countably infinite set of node variables, ranged over by $X, Y, \ldots$. To evaluate a parallel map operation, we need to regard a system of equations as a finite map from a finite domain of atomic elements to values. For this purpose, we also assume that there is a countably infinite set of *node atoms* $A_i^j (1 \leqslant i \leqslant j)$, whose finite subset is represented by a node type variable *n*.

The set of semantic values, ranged over by V, for the language is given by the following syntax

$V ::=$	c	(constant)
$\mid$	$*$	(unit)
$\mid$	$\mathtt{fcl}(\rho, f, x, M)$	(function closure)
$\mid$	(V, V)	(pair)
$\mid$	$inl(V)$	(left injection)
$\mid$	$inr(V)$	(right injection)
$\mid$	$A_i^j \ (1 \leqslant i \leqslant j)$	(node atom)
$\mid$	X	(node variable)
$\mid$	$X.\langle X_1 \to V_1, \ldots, X_k \to V_k \rangle$	(system of equations)
$\mid$	$\mathtt{fmap}(A_1^k \to V_1, \ldots, A_k^k \to V_k)$	(finite map function)

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

where ρ stands for a run-time environment, which is a finite function from identifiers to values. As explained before, the sequence $X.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle$ is an association sequence, which is a run-time representation of a system of equations. In an association sequence, the set of node variables $X_i, \ldots, X_k$ are pairwise distinct and the order of the associations is insignificant. Association sequences are often denoted by $\chi, \chi', \ldots$. The concatenation of two sequences, denoted as $\chi \mathbin{@} \chi'$, is allowed only when the sets of node variables constituting the sequences are disjoint. Concatenation of multiple sequences $\chi_1 \mathbin{@} \cdots \mathbin{@} \chi_k$ is abbreviated as $@_{i=1}^{k}\ \chi_i$. $\mathtt{fmap}(A_1^k \to V_1, \cdots, A_k^k \to V_k)$ represents a semantic finite map function from the set of node atoms $\{A_1^k, \ldots, A_k^k\}$ to values. The set of node atoms are classified by the super-scripted number: the node atoms $A_1^k, \ldots, A_k^k$ are used to constitute the semantic finite map function whose domain consists of k node atoms. These node atoms are introduced each time a parallel map is applied to a system of equations in order to set up the special functions bound by the identifiers f and d, where the set of node atoms $\{A_1^k, \ldots, A_k^k\}$ is selected for the temporary names of the k node variables constituting the given system of equations.

The node variables $X_1, \ldots, X_k$ in an association sequence $X_i.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle$ are bound variables. An occurrence of a node variable X in V is *free*, if it is not contained in any association sequence of the form $X_i.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle$ such that $X = X_j$ for some j; it is *bound* otherwise. We identify association sequences that differ only in the names of the set of bound node variables, and further adopt the "bound variable conventions" for bound node variables. We define a substitution for free node variables to be a finite function from node variables to values, written as $[X_1 \mapsto V_1, \ldots, X_k \mapsto V_k]$. Similar to type substitutions, a substitution S for node variables is extended to the substitution for values by the induction on the structure of values. The result of applying S to an association sequence $X.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle$ is the association sequence obtained by applying S to its all *free node variables*. Under the bound node variable convention, we can simply take $S(X.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle) = X.\langle X_1 \to S(V_1), \ldots, X_k \to S(V_k)\rangle$. The result of applying a substitution S to a function closure $\mathtt{fcl}(\rho, f, x, M)$ is the function closure $\mathtt{fcl}(S(\rho), f, x, M)$, where $S(\rho)$ is defined to be a run-time environment such that $Dom(S(\rho)) = Dom(\rho)$ and $S(\rho)(x) = S(\rho(x))$ for all $x \in Dom(S(\rho))$.

4.2.2 Semantic rules

We give the semantics by a set of rules that define the relation

$$\rho \vdash M \Downarrow r,$$

representing the fact that the expression M evaluates to a result r under the environment ρ. The set of results is defined by the following grammar:

$$r ::= V \mid \mathit{wrong}$$

where *wrong* represents the run-time error. The set of rules for the language is given in Table 2, where we assume that the rules yielding *wrong* is implicitly given: a derivation yields *wrong*, if the results of some sub-derivations do not match the pattern of the values in any rules, or some premises (such as $x \in Dom(\rho)$) are not satisfied.

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

Table 2. *Parallel operational semantics*

constant

$$\rho \vdash c \Downarrow c$$

identifier

$$\frac{x \in Dom(\rho)}{\rho \vdash x \Downarrow \rho(x)}$$

function

$$\rho \vdash fun\ f(x) \Rightarrow M \Downarrow \mathtt{fcl}(\rho, f, x, M)$$

function application

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

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

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

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

let

$$\frac{\rho \vdash M \Downarrow V' \quad \rho + \{x \mapsto V'\} \vdash N \Downarrow V}{\rho \vdash let\ x = M\ in\ N \Downarrow V}$$

foreach

$$\frac{\rho \vdash M \Downarrow X_j.\langle X_1 \to V_1', \cdots, X_k \to V_k' \rangle \quad \rho + \left\{ \begin{array}{l} f \mapsto \mathtt{fmap}(A_1^k \to Y_1, \ldots, A_k^k \to Y_k), \\ d \mapsto \mathtt{fmap}(A_1^k \to S(V_1'), \ldots, A_k^k \to S(V_k')), \\ x \mapsto S(V_i') \end{array} \right\} \vdash N \Downarrow V_i \quad \text{for each } i, 1 \leqslant i \leqslant k}{\rho \vdash foreach\ x \in M\ with\ [f, d]\ do\ N \Downarrow Y_j.\langle Y_1 \to V_1, \cdots, Y_k \to V_k \rangle}$$

($Y_1, \ldots, Y_k$ are fresh node variables, and $S = [X_1 \mapsto A_1^k, \ldots, X_k \mapsto A_k^k]$.)

fmap application

$$\frac{\rho \vdash M \Downarrow \mathtt{fmap}(A_1^k \to V_1, \ldots, A_k^k \to V_k) \quad \rho \vdash N \Downarrow A_i^k}{\rho \vdash MN \Downarrow V_i}$$

up

$$\frac{\rho \vdash N_i \Downarrow X_i.\chi_i \quad \text{for each } i, 1 \leqslant i \leqslant m \quad \rho + \{x_1 \mapsto X_1, \ldots, x_m \mapsto X_m\} \vdash M \Downarrow V}{\rho \vdash up\ M\ with\ x_1 = N_1, \ldots, x_k = N_m\ end \Downarrow X.(\langle X \to V \rangle\ @\ (@_{i=1}^m\ \chi_i))}$$

(X is a new node variable.)

dn

$$\frac{\rho \vdash M \Downarrow X_i.\langle X_1 \to V_1, \cdots, X_k \to V_k \rangle}{\rho \vdash dn(M) \Downarrow S(V_i)}$$

($S = [X_1 \mapsto X_1.\chi, \ldots, X_k \mapsto X_k.\chi]$ where $\chi = \langle X_1 \to V_1, \ldots, X_k \to V_k \rangle$])

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

In the rule for the *foreach* expression, M is first evaluated to an association sequence $X_j.\langle X_1 \to V'_1, \ldots, X_k \to V'_k\rangle$. Then the set of node atoms $\{A^k_1, \ldots, A^k_k\}$ matching the length of the sequence is introduced as the set of temporary names. The map operation $\lambda x.N$ is applied to each $S(V'_i)$ where S is the substitution which renames each node variable X_i to the corresponding node atom A^k_i. The special functions f and d are bound to finite maps from the node atoms $A^k_1, \ldots, A^k_k$ to values. The semantics for the finite function application is given in the table in an obvious way. The renaming function f is a one-to-one finite map from the node atoms to fresh node variables $Y_1, \ldots, Y_k$. The down function d maps each node atom A^k_i to its corresponding value $S(V'_i)$. The final result is obtained by associating each fresh node variable Y_i to the value V_i, the corresponding result of map operation. The root node is the node variable Y_j which corresponds to the root node of the original system of equations.

The rule for *up* M *with* $x_1 = N_1, \ldots, x_m = N_m$ *end* first evaluates each sub-data N_i to a system of equations $X_i.\chi_i$. Then M is evaluated to a value V under the binding $\{x_1 \mapsto X_1, \ldots, x_k \mapsto X_k\}$. The final result is the association sequence $X.\langle X \to V\rangle \ @ \ (@^m_{i=1}\ \chi_i)$ where X is a new node variable. We can assume that the set of node variables constituting each association sequence χ_i is disjoint by renaming node variables. The rule for $dn(M)$ first evaluates M to yield a system of equations $X_i.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle$. Then it returns the value obtained from V_i by instantiating its immediate sub-nodes, i.e. by replacing each free occurrence of X_j in V_i with the corresponding system of equations, $X_j.\langle X_1 \to V_1, \cdots, X_k \to V_k\rangle$.

4.3 Type soundness

We show that the presented type system is sound with respect to the operational semantics. There is one subtlety in establishing this desired property; the evaluation of parallel maps involves application of finite maps, and the type system must also ensure that a finite map always apply to the elements of its domain. For the purpose of proving the fact, we assume that each node type variable is assigned a fixed natural number k describing the cardinality of the corresponding set of node atoms $\{A^k_1, \ldots, A^k_k\}$. We write $|\, n \,|$ to denote the cardinality number assigned to a node variable n, and we assume that there are countably infinite node type variables n such that $|\, n \,| = k$.

We prove the type soundness in the style of Leroy (1992), which refines the treatment of store typing in Tofte's (1988) proof. Types of semantic values are defined relative to a *recursive type variable environment*, which is similar to Leroy's store typing. A recursive type variable environment $\mathscr{R}$ is a finite map from recursive type variables to the sets of node variables, written as

$$\mathscr{R} = \{t_1 \mapsto \{X^1_1, \ldots, X^1_{n(1)}\}, \ldots, t_k \mapsto \{X^k_1, \ldots, X^k_{n(k)}\}\}.$$

The type system of values is defined as a set of rules to derive the following form of judgment

$$\mathscr{R} \models V : \tau$$

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

denoting the fact that the value V has the type τ under the type variable environment $\mathscr{R}$. The typing rules for values are defined below.

- $\mathscr{R} \models c : \tau$ if constant c has type τ.
- $\mathscr{R} \models A_i^j : n$ if $\mid n \mid = j$
- $\mathscr{R} \models \mathtt{fcl}(\rho, f, x, M) : \tau_1 \to \tau_2$ if there exists Γ such that $\mathscr{R} \models \rho : \Gamma$ and $\Gamma \vdash \mathit{fun}\ f(x) \Rightarrow M : \tau_1 \to \tau_2$.
- $\mathscr{R} \models (V_1, V_2) : \tau_1 \times \tau_2$ if $\mathscr{R} \models V_1 : \tau_1$ and $\mathscr{R} \models V_2 : \tau_2$.
- $\mathscr{R} \models \mathit{inl}(V) : \tau_1 + \tau_2$ if $\mathscr{R} \models V : \tau_1$.
- $\mathscr{R} \models \mathit{inr}(V) : \tau_1 + \tau_2$ if $\mathscr{R} \models V : \tau_2$.
- $\mathscr{R} \models X : t$ if $t \in \mathit{Dom}(\mathscr{R})$ and $X \in \mathscr{R}(t)$.
- $\mathscr{R} \models X.\langle X_1 \to V_1, \cdots, X_k \to V_k \rangle : \mu t.\tau$ if $X_1, \ldots, X_k$ are distinct, $X = X_i$ for some $i (1 \leqslant i \leqslant k)$, and $\mathscr{R} + \{t \mapsto \{X_1, \ldots, X_k\}\} \models V_i : \tau$ holds for all $i (1 \leqslant i \leqslant k)$. (We assume $\{X_1, \ldots, X_k\} \cap \mathscr{R}(s) = \emptyset$ for all $s \in \mathit{Dom}(\mathscr{R})$ by the renaming convention.)
- $\mathscr{R} \models \mathtt{fmap}(A_1^k \to V_1, \cdots, A_k^k \to V_k) : n \to \tau$ if $\mid n \mid = k$ and $\mathscr{R} \models V_i : \tau$ for all $i (1 \leqslant i \leqslant k)$.
- $\mathscr{R} \models V : \sigma$ if $\mathscr{R} \models V : \tau$ for all τ such that $\tau \preceq \sigma$.
- $\mathscr{R} \models \rho : \Gamma$ if $\mathit{Dom}(\rho) = \mathit{Dom}(\Gamma)$ and $\mathscr{R} \models \rho(x) : \Gamma(x)$ for all $x \in \mathit{Dom}(\rho)$.

The following theorem shows the type system is sound with respect to the operational semantics, i.e. no well-typed program causes run-time error.

Theorem 1 (*Type soundness*)
Let Γ be a type environment, M be an expression, and τ be a type such that $\Gamma \vdash M : \tau$. Then, for any $\mathscr{R}$ and ρ such that $\mathscr{R} \models \rho : \Gamma$ and $\rho \vdash M \Downarrow r$, we have $r \neq \mathit{wrong}$ and $\mathscr{R} \models r : \tau$.

The proof of this theorem is deferred until Appendix A.

5 Examples of data-parallel programming

The language we have just defined is intended to serve as a core language for data-parallel programming with general recursive types. However, the raw syntax is too verbose. For ease of programming with recursive types, we introduce data type declarations and a high level syntax similar to that employed by existing ML languages. A data type declaration defines the structure of a recursive datum and the data constructor names. For example, list type is defined as follows:

```
datatype α list = Nil | Cons of α * α list;
```

As in ML, this defines type `α list` of lists of elements of type `α`, and data constructors `Nil` and `Cons` for lists. The constructor names are also used as patterns in pattern matching constructs such as *case* and *let* expressions. Different from the conventional functional languages, however, we should be able to use a constructor name in two different meanings: one for folding or unfolding a recursive datum represented as a system of equations; and the other for expressing node values in a system of equations. These two are distinguished in our high level syntax by a

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

prefix. If '^' is prefixed to a constructor name, it expresses a recursive datum of some type $\mu t.\tau(t)$; otherwise, it is just a node value of type $\tau(t)$. For example, a sequential version of `map` function is implemented as follows:

```
fun map F L =
    case L of
      ^Nil => ^Nil
    | ^Cons(hd,tl) => ^Cons(F hd, map F tl)
    end;
```

where the expressions `^Nil` and `^Cons(F hd, map F tl)` on the right-hand of the case arrows construct lists whose type is a recursive type $\mu t.unit + \alpha \times t$, and the patterns `^Nil` and `^Cons(hd,tl)` are used to match a value of the same recursive type. The counterpart of this program in the raw syntax is given below.

$$\begin{array}{l} fun\ map(F) \Rightarrow \lambda L.case\ dn(L)\ of \\ \qquad\qquad inl(x) \Rightarrow up(inl(*)), \\ \qquad\qquad inr(y) \Rightarrow let\ hd = fst(y) \\ \qquad\qquad\qquad\qquad in\ let\ tl = snd(y) \\ \qquad\qquad\qquad\qquad in\ up\ inr((F\,hd, z))\ with\ z = map\ F\ tl\ end \end{array}$$

The data-parallel version of `map`, which we have described in the raw syntax in section 3.2.1, is written with the unprefixed constructor names as follows:

```
fun map F L =
    foreach x in L with [f,d] do
        case x of
          Nil => Nil
        | Cons(hd,tl) => Cons(F hd, f tl)
        end
    end;
```

where `Nil` and `Cons` are used as constructors and patterns for node values of type $unit + \alpha \times t$.

5.1 Data-parallel suffix sum via data-parallel recursion

We show how the data-parallel list suffix algorithm given in section 2 is expressed in our language. The function `suffix` in Figure 4 takes two arguments, an associative binary operator `op` and a list `L`, and performs the parallel suffix operation. This function has a polymorphic type $\forall\alpha.(\alpha \to \alpha \to \alpha) \to (\mu t.unit + \alpha \times t) \to (\mu t.unit + \alpha \times t)$. Therefore, the function is a general parallel suffix operation that works for any list and any associative binary operation on the list elements. For example, if `op` is the integer sum and `L` is an integer list, the function `suffix` returns the suffix sum of the integer list.

The diagram in Figure 5 illustrates how `suffix` works. The function `suffix` is a composite of three functions `mk_chum`, `scan`, and `strip`. First `mk_chum`, a parallel map function, transforms the given list to the list with chums of type $\mu t.unit + \alpha \times t \times t$. (The list with `chum` field is represented in the program by the data type α `cList`.)

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

```
datatype α List = Nil | Cons of α * α List;
datatype α cList = cNil | cCons of α * α cList * α cList;

fun suffix op L =
let fun mk_chum L =
        foreach x in L with [f,d] do
            case x of
              Nil => cNil
            | Cons(hd,tl) => cCons(hd,f tl, f tl)
            end
        end
    fun square cL =
        foreach x in cL with [f,d] do
            case x of
              cNil => cNil
            | cCons(n,chum,tl) =>
                case (d chum) of
                  Nil => cCons(n,f chum,f tl)
                | cCons(m,chum',_) => cCons((op n m),f chum',f tl)
                end
            end
        end
    fun strip cL =
        foreach x in cL with [f,d] do
            case x of
              cNil => Nil
            | cCons(n,chum,tl) => Cons(n,f tl)
            end
        end
    fun scan cL =
        case cL of
          ^cNil => ^cNil
        | ^cCons(_,chum,_) =>
            case chum of
              ^cNil => cL
            | ^cCons(_,_,_) => scan (square cL)
            end
        end
in
    strip (scan (mk_chum L))
end;
```

Fig. 4. Data-parallel suffix program.

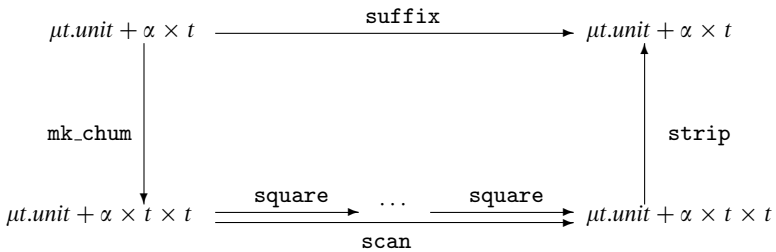

Fig. 5. Diagram for `suffix`.

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

The suffix of the list is computed by the function `scan`, a function for data-parallel recursion, which iterates the function `square` repeatedly until the chum pointer in the first cons cell reaches `cNil`. Finally, the function `strip` is applied to transform the list with `chum` fields into the ordinary list.

In a similar way, we can describe data-parallel algorithms based on pointer jumping technique. In Appendix B, a more complex data-parallel programming example for binary tree is given. The program describes the algorithm called the parallel pebble game, a variant of the so-called tree contraction algorithm, which takes $O(\log n)$ parallel steps to perform suffix computation on n node binary trees of arbitrary shape.

As we noted earlier, the purpose of our language is to express the essential nature of data-parallel algorithms in a typed higher-order functional language. The above examples demonstrate that data-parallel algorithms based on pointer jumping technique are cleanly represented. Of course, the pointer jumping technique is *not* the only one technique. There are several other significant programming techniques, and some of them may not be easily expressed in our framework. To write a program in our language that solves a particular problem, we need to find a proper data-parallel algorithm that can be expressed in our framework, which is out of the scope of the present paper.

6 A parallel execution model for the language

This section describes the intended parallel execution model of our language that respects the formal semantics given in the previous section. For the purpose of giving the principal idea of the parallel execution, we only consider a rather simplified case:

1. We assume SPMD execution on an idealized distributed memory multicomputer which consists of an unbounded number of processors. Each processor in the idealized multicomputer has a local memory and is identified by a unique processor id. We also assume that the multicomputer provides the facilities for broadcasting and inter-processor communication.
2. We do not consider any nested parallelism, i.e. nested execution of *foreach*, *up*, and *dn*.
3. We only consider simple recursive data structures such as integer lists and binary trees.

Apparently, the parallel execution model we shall describe does not support the full specification of the language and that it does not immediately yields a practical compilation method either. Some important points that remain to be considered includes the following:

1. In practice, the number of processors is bounded and possibly very small. We need a method to allocate a limited number of physical processors to parallel operations.
2. To support nested parallelism, we need to map the execution of the nested expressions onto a flat parallel architecture. One promising way for doing

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

this is *flattening*, which is a transformation technique for unnesting the nested expressions. There are several proposals to flatten arrays (Blelloch, 1990; Prins and Palmer, 1993; Keller and Simons, 1996). Applying these methods to our language requires further refinement.

3. The representation of nested recursive data, which would be closely related to the nested parallelism, is not obvious. Even for a simple recursive data, a subtle problem arises due to sharing of sub-structures. For example, consider constructing a binary tree from two different subtrees. If some of data elements of the both subtrees reside in the same virtual processors, then we need to allocate new virtual processors for the duplicated data elements. Functional recursive types such as $\mu t.unit + int \to t$ add more complication since function closures are passed around among the processors.
4. Data alignment is a critical issue for tuning the performance of parallel computation: data elements should be placed so that the communication overhead becomes as small as possible under the network configuration of a specific architecture. This is more difficult for our language which can dynamically allocate recursive data, while data alignment is optimized usually due to the static size information of arrays.

As indicated by these points, significant further research will be needed to make the parallel execution model to be a practical implementation method for our language on an actual parallel machine. Furthermore, since our intended parallel execution model is based on SPMD, which basically assumes a fine grain parallel architecture, the proposed method may not be suitable for currently popular parallel architectures of coarser grain. We nonetheless believe that our execution model is conceptually worth considering for the study of data-parallelism on recursive data: the model could be a basis of a general SPMD execution method for data-parallel programs with arbitrary recursive data that are described by means of what we call parallel recursion.

6.1 Representing recursive data by association pointers

As explained in section 3, the run-time value of a recursive datum is an association sequence $X_j.\langle X_1 \to V_1, \ldots, X_n \to V_n\rangle$ representing a system of equations. The intention of this representation is that the associated values $V_1, \ldots, V_n$ are distributed over distinct processors identified by the node variables $X_1, \ldots, X_n$, respectively. Using processor id's as node variables, we map an association sequence $X_j.\langle X_1 \to V_1, \ldots, X_n \to V_n\rangle$ onto the multicomputer as follows. Suppose that the associations $X_1 \to V_1$, …, $X_n \to V_n$ are allocated to the processors P_1, …, P_n, respectively. By this distribution, the associations are localized in each processor, but the information on root node must be globally shared by all the processors. We therefore represent the association sequence in each processor P_i as an *association pointer* $A_{P_i} = (P_j, p_{P_i})$, which is a pair of a processor id and a pointer to a local datum. In all the processors, the association pointer has the same processor id P_j as the first element, which indicates the processor allocating the root node. The second element p_{P_i} is a pointer referring to the allocated node value stored in the local

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

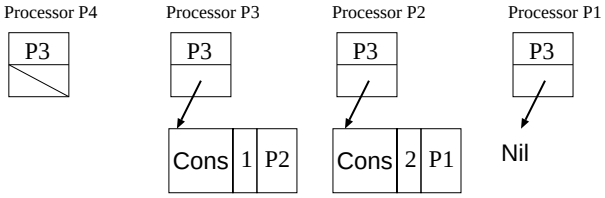

Fig. 6. Representation of a list by association pointers.

memory of each processor P_i. The processors that are not allocated an association has an association pointer $A = (P_j, \mathit{Undefined})$, where the special value *Undefined* indicates that the processor is not in charge of processing the system of equations. Figure 6 shows how a list [1, 2] represented by the following run-time representation

$$X_1.\langle X_1 \rightarrow \mathit{inr}((1, X_2)), X_2 \rightarrow \mathit{inr}((2, X_3)), X_3 \rightarrow \mathit{inl}(*)\rangle$$

is expressed by association pointers. In the figure, an association pointer is depicted by two vertically stacked boxes, where the top box represents the processor id and the bottom one represents the pointer to the allocated cons cell or `Nil`. The special value *Undefined* is represented in the figure by a backslash. Each cons cell contains a processor id in its cdr part to represent the corresponding node variable indexing the processor that is allocated the next cons cell.

6.2 Execution of parallel map

A parallel map *foreach* $x \in N$ *with* $[f, d]$ *do* M is evaluated by simultaneously applying the map function $\lambda x.M$ to the value allocated in each processor. The instruction stream executed in each processor is dependent on the allocated value, and inter-processor communication is triggered by the application of the down function d.

Suppose N is evaluated to a recursive datum, and it is represented in every processor P by the association pointers $A_P = (P^r, p_P)$. First, every processor P that is not allocated a node value (i.e. if $p_P = \mathit{Undefined}$) become inactive. Then, in each remaining active processor P, the variable f is bound to the identity function and the variable d is bound to a function which, given a processor id P, fetches the value allocated to processor P via inter-processor communication. Under these bindings, the variable x is bound to the value stored in the local address p_P, and M is evaluated simultaneously in each processor. The final result in each active processor is an association pointer (P^r, p) where p is the pointer to the value obtained by evaluating M. Execution of the parallel map is finished by resuming the execution of the inactivated processors. As a simple example, consider the following parallel map:

$$\begin{array}{l} \mathit{foreach}\ x \in M\ \mathit{with}\ [f, d]\ \mathit{do} \\ \quad \mathit{case}\ x\ \mathit{of}\ \mathit{inl}(y) \Rightarrow \mathit{inl}(y), \\ \qquad\qquad \mathit{inr}(y) \Rightarrow \mathit{case}\ (d\, \mathit{snd}(y))\ \mathit{of}\ \mathit{inl}(z) \Rightarrow \mathit{inr}((\mathit{fst}(y), f\ \mathit{snd}(y))) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{inr}(z) \Rightarrow \mathit{inr}((\mathit{fst}(y) + \mathit{fst}(z), f\ \mathit{snd}(y))) \end{array}$$

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

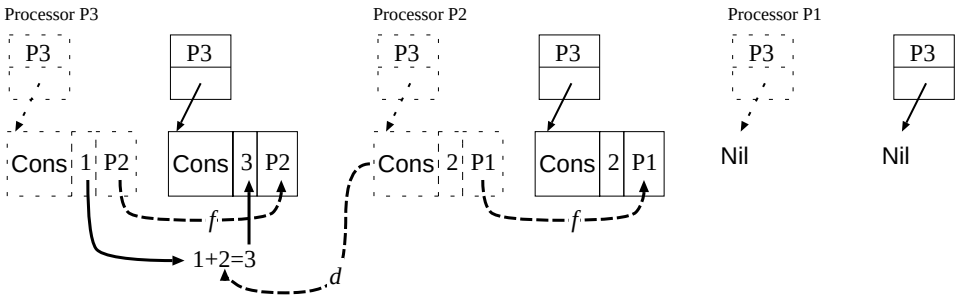

Fig. 7. Execution of *foreach* on a list.

This is a parallel version of the following sequential recursive function:

```
fun foo L = case L of
              ^Nil => ^Nil
            | ^Cons(n,tl) => case tl of
                               ^Nil => ^Cons(n,foo tl)
                             | ^Cons(m,_) => ^Cons(n+m,foo tl)
                             end
            end;
```

We illustrate how the parallel map is executed in Figure 7 in the case $M = [1, 2]$. In the figure, the old data is drawn by dashed thin lines, and the resulting data is drawn by plain thin lines. The dashed thick arrows labeled by f represent copying of a processor id, the dashed thick arrow labeled by d represents data transfer by inter-processor communication, and the thick arrows represent data flow inside a processor.

6.3 *Execution of* up *and* dn

To construct a recursive datum by *up* M *with* $x_1 = N_1, \ldots, x_m = N_m$ *end*, every processor P first evaluates each N_i to an association pointer $A_P = (P^{r_i}, p_P)$ and M to a new value under the binding of each x_i to the corresponding association pointer. Then every processor selects the same processor Q to which any elements of recursive data have not been allocated. Then the processor Q yields an association pointer (Q, q) where q is the pointer to a value obtained by replacing every association pointer (P', p) contained in the value evaluated from M by the processor id P'. Every processor P other than Q discards the value evaluated from M and yields an association pointer (Q, p_P). Figure 8 illustrates how *up* $inr((0, x))$ *with* $x = N$ *end*, i.e. a cons operation, is executed and yields an association pointer in each processor, in the case $N = [1, 2]$.

A recursive data M is destructed by $dn(M)$ as follows. First, every processor P evaluates M to obtain an association pointer $A_P = (P^r, p_P)$. The processor P^r broadcasts the node value allocated to the processor to all the processors. Then every

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

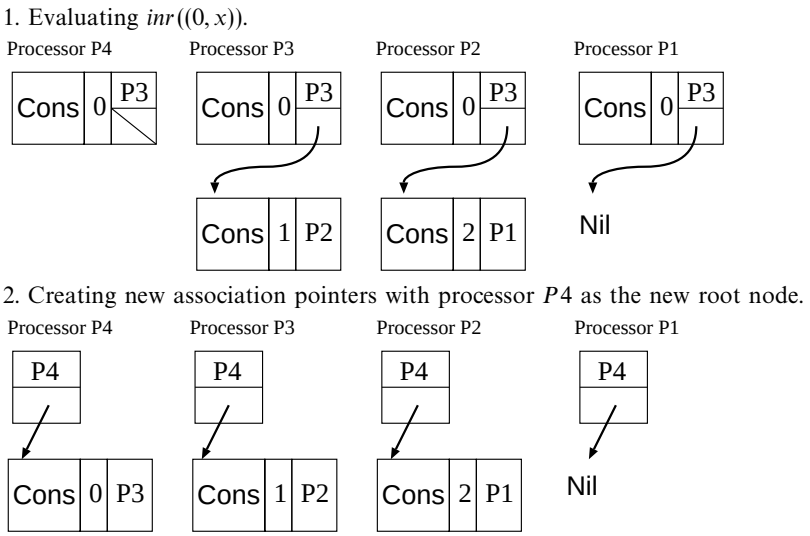

Fig. 8. Execution of *Cons* on a list.

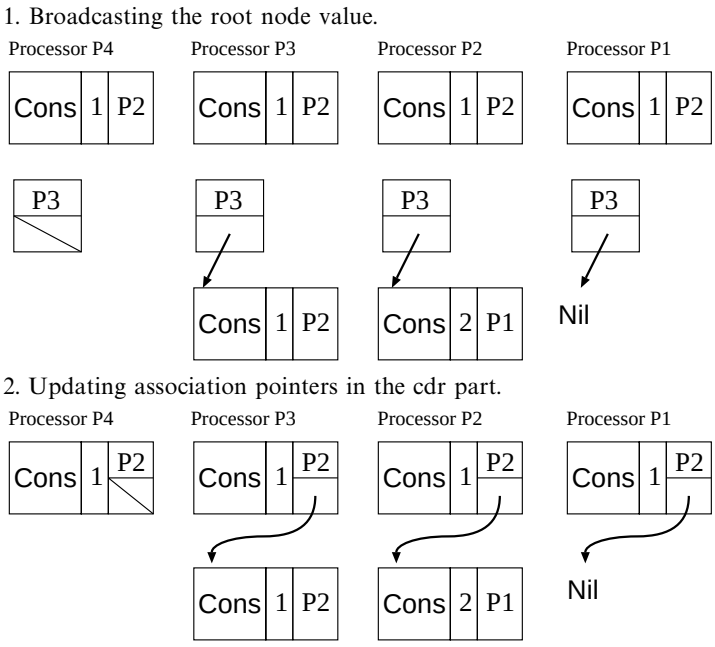

Fig. 9. Application of *dn* to a list.

processor P other than the processor P^r replace every processor id Q contained in the broadcasted value by an association pointer (Q, p_P). The processor P^r yields the node value that has been allocated to the processor. Figure 9 shows how $dn(M)$ is executed for $M = [1, 2]$.

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

7 Conclusion

We have proposed a new language mechanism, called data-parallel recursion, for data-parallel processing of recursive data, and have worked out a functional language suitable for describing data-parallel programs via data-parallel recursion. The language supports distributed recursive data and a data-parallel function application mechanism, called parallel map. The combination of the parallel map and the ordinary recursion allows us to describe a class of typical data-parallel algorithms on recursive data in a uniform way. Furthermore, the language is a typed functional language with ML polymorphic let and the type system is sound with respect to a formal operational semantics. In addition to the formal operational semantics, we have also given a simple data-parallel execution model for the language.

To substantiate the feasibility of the data-parallel execution model, the authors designed a prototype ML like data-parallel language embodying the results presented in this paper, and implemented a prototype system on a SIMD architecture with 1024 PEs. The prototype system is a translator that converts the prototype language into TUPLE language (Yuasa, 1992), a data-parallel variant of Lisp language. The translator type checks a program and then generates the corresponding target Lisp code for each data-parallel construct.

In a wider perspective, the paradigm of parallel transformation of system of equations can be applied not only to data-parallel programming on recursively defined data but also to various other areas where manipulation of mutually dependent structures are crucial, such as object-oriented databases. Based on this general observation, we have recently proposed an object-oriented data model and a data-parallel query language (Nishimura *et al.*, 1996). In this model, the idea of parallel map not only contributes to parallel query processing but it also enables us to achieve declarative database query involving object identity, which have traditionally been processed in an *ad hoc* way using pointer manipulation.

Acknowledgments

We thank Taiichi Yuasa for allowing us to use the TUPLE language system on a parallel machine with 1024 PEs, which enabled us to implement a prototype data-parallel language described in this article. We also thank the anonymous reviewers whose comments were very helpful to improve the paper.

A Proof of type soundness

This appendix gives a proof for the soundness theorem. The proof structure is similar to Leroy's (1992).

Proposition 2
Let Γ be a type environment, M be an expression, and τ be a type such that $\Gamma \vdash M : \tau$. Then, the following properties hold:

1. For any Γ' and τ' obtained by replacing every occurrence of a node type

variable n in Γ and τ by some fresh node type variable n', there is a derivation such that $\Gamma' \vdash M : \tau'$.

2. For any Γ' and τ' obtained by replacing every free occurrence of a recursive type variable t in Γ and τ by some type τ'', there is a derivation such that $\Gamma' \vdash M : \tau'$.
3. For any substitution θ for polymorphic type variables, there is a derivation such that $\theta(\Gamma) \vdash M : \theta(\tau)$, where $\theta(\Gamma)$ represents substitution for type environment, i.e. $Dom(\theta(\Gamma)) = Dom(\Gamma)$ and $\theta(\Gamma)(x) = \theta(\Gamma(x))$ for all $x \in Dom(\Gamma)$.

Proof

Properties 1 and 2 Easy induction on the structure of M.

Property 3 The proof is induction on the structure of M. The proof is the same as the Leroy's (1992), except for the case M is either *up*, *dn*, or *foreach*. We only prove the case *foreach*. (The other two cases are proved in a similar way.)

Case $M = foreach\ x \in M\ with\ [f,d]\ do\ N$.

The type of M is derived as follows.

$$\frac{\begin{array}{c}\Gamma \vdash M : \mu t.\tau(t) \\ \Gamma + \{f : n \to s, d : n \to \tau(n), x : \tau(n)\} \vdash M : \tau'(s) \\ n \text{ does not occur free in } \tau'(s)\end{array}}{\Gamma \vdash foreach\ x \in M\ with\ [f,d]\ do\ N : \mu t.\tau'(t)}$$

We can assume that n does not occur free in $\theta(\alpha)$ for all $\alpha \in Dom(\theta)$, by renaming the node type variable n to a fresh one using property 1 if necessary. By the induction hypothesis, we have the following derivations:

$$\theta(\Gamma) \vdash M : \mu t.\theta(\tau(t)) \text{ and}$$
$$\theta(\Gamma) + \{f : n \to s, d : n \to \theta(\tau(n)), x : \theta(\tau(n))\} \vdash M : \theta(\tau'(s)).$$

By the assumption, n does not occur free in $\theta(\tau'(s))$ either. As a consequence, we can conclude

$$\theta(\Gamma) \vdash foreach\ x \in M\ with\ [f,d]\ do\ N : \theta(\mu t.\tau'(t)).$$

□

The following lemma shows that value typing is stable under extensions of the recursive type variable environment.

Lemma 3

Let $\mathscr{R}$ be a type variable environment, V be a semantic value, and τ be a type such that $\mathscr{R} \models V : \tau$. Then, the relation $\mathscr{R}' \models V : \tau$ holds for any recursive type environment $\mathscr{R}'$ such that $Dom(\mathscr{R}') \supseteq Dom(\mathscr{R})$ and $\mathscr{R}'(t) \supseteq \mathscr{R}(t)$ for all $t \in Dom(\mathscr{R})$.

Proof

By induction on the structure of V. □

The next lemma shows that the type $\tau(t)$ of a value V is changed to $\tau(\tau')$ if every node variable in V that belongs to type t is replaced by a value of type τ'.

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

Lemma 4
Let $\mathscr{R}$ be a type variable environment, V be a semantic value, $X_1, \ldots, X_k$ be node variables, and $\tau(t)$ be a type such that $\{X_1, \ldots, X_k\} \cap \mathscr{R}(s) = \emptyset$ for all $s \in Dom(\mathscr{R})$ and $\mathscr{R} + \{t \mapsto \{X_1, \ldots, X_k\}\} \models V : \tau(t)$. Suppose S be a substitution for node variables such that $Dom(S) = \{X_1, \ldots, X_k\}$ and $\mathscr{R} \models S(X_i) : \tau'$ for all $i (1 \leqslant i \leqslant k)$, where τ' does not contain any free occurrence of t. Then, the relation $\mathscr{R} \models S(V) : \tau(\tau')$ holds.

Proof
Let $\mathscr{R}' = \mathscr{R} + \{t \mapsto \{X_1, \ldots, X_k\}\}$. Proof is induction on the structure of V. We prove only one crucial case.

Case $V = Y_j.\langle Y_1 \to V_1, \ldots, Y_k \to V_k \rangle$**.**
We can assume that $\tau(t) = \mu s.\tau''(t)$ for some $\tau''(t)$ and that $t \neq s$ by the renaming convention. Hence, by the definition of value typing and by lemma 3, we have $\mathscr{R}' + \{s \mapsto \{Y_1, \ldots, Y_k\}\} \models V_i : \tau''(t)$ for all i. By applying the induction hypothesis, we have $\mathscr{R}' + \{s \mapsto \{Y_1, \ldots, Y_k\} \models S(V_i) : \tau''(\tau')$ for all i, and therefore $\mathscr{R} \models S(V) : \tau(\tau')$.

□

We show in the following proposition that the value typing is stable under substitutions for polymorphic type variables.

Proposition 5
Let $\mathscr{R}$ be a type environment, V be a semantic value, and τ be a type such that $\mathscr{R} \models V : \tau$. Then, for any polymorphic type substitution θ, the relation $\mathscr{R} \models V : \theta(\tau)$ holds. As a consequence, we have $\mathscr{R} \models \forall \alpha_1 \cdots \alpha_k.\tau$ for any set of polymorphic type variables $\alpha_1, \ldots, \alpha_k$.

Proof
The proof is induction on the structure of V. Most of the induction cases are easy to prove, except for the case V is a function closure.

Case $M = \texttt{fcl}(\rho, f, x, M)$ *and* $\tau = \tau_1 \to \tau_2$**.**
Let Γ be a type environment such $\mathscr{R} \models \rho : \Gamma$ and $\Gamma \vdash \texttt{fcl}(\rho, f, x, M) : \tau_1 \to \tau_2$. By proposition 2(property 3), we have

$$\theta(\Gamma) \vdash \texttt{fcl}(\rho, f, x, M) : \theta(\tau_1 \to \tau_2).$$

To show $\mathscr{R} \models \rho : \theta(\Gamma)$, let $\Gamma(y) = \forall \alpha_1 \cdots \alpha_n.\tau_y$ for any $y \in Dom(\rho)$. By the renaming convention, we have $\theta(\Gamma(y)) = \forall \alpha_1 \cdots \alpha_n.\theta(\tau_y)$. We need to show that $\mathscr{R} \models \rho(y) : \tau'$ for any τ' such that $\tau' \preceq \theta(\Gamma(y))$. There exists a type substitution ψ such that $\tau' = \psi(\theta(\tau_y))$ for any instance τ'. From the definition of value typing for type schemes, we also have $\mathscr{R} \models \rho(y) : \tau_y$. Therefore, by the induction hypothesis, we have $\mathscr{R} \models \rho(y) : \tau'$. Since ψ is arbitrary, we have $\mathscr{R} \models \rho(y) : \theta(\Gamma(y))$. This holds for all $y \in Dom(\Gamma)$. Therefore we can conclude $\mathscr{R} \models \rho : \theta(\Gamma)$.

□

We are now ready to prove the type soundness theorem.

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

Proof
The proof is by induction on the height of the derivation tree $\rho \vdash M \Downarrow V$. We prove according to the last rule used to derive the value V. In the following, we prove some non-trivial cases only.

Case $M = let\ x = M\ in\ N$**.**
The type is derived as follows:

$$\frac{\Gamma \vdash M : \tau' \quad \Gamma + \{x : Clos_\Gamma \tau'\} \vdash N : \tau}{\Gamma \vdash let\ x = M\ in\ N : \tau}$$

By the induction hypothesis, we have

$$\rho \vdash M \Downarrow V' \text{ and } \mathscr{R} \models V' : \tau'.$$

By proposition 5, we have $\mathscr{R} \models V' : Clos_\Gamma \tau'$. By applying the induction hypothesis, we obtain the relations:

$$\rho + \{x \mapsto V'\} \vdash N \Downarrow V \text{ and } \mathscr{R} \models V : \tau,$$

and we have V as the result of evaluation by the following evaluation derivation:

$$\frac{\rho \vdash M \Downarrow V' \quad \rho + \{x \mapsto V'\} \vdash N \Downarrow V}{\rho \vdash let\ x = M\ in\ N \Downarrow V}$$

Case $M = MN$**.**
The type is derived as follows.

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

There are two possible evaluation derivations corresponding to this type derivation.

Case *M is evaluated to a functional closure*.
By the induction hypothesis, we have $\mathscr{R} \models \mathtt{fcl}(\rho', f, x, M') : \tau_1 \to \tau_2$ and $\mathscr{R} \models V' : \tau_1$. By the definition of functional closure typing, there exists a type environment Γ' such that $\mathscr{R} \models \rho' : \Gamma'$ and $\Gamma' + \{f : \tau_1 \to \tau_2, x : \tau_1\} \vdash M' : \tau_2$. As $\mathscr{R} \models \rho' + \{f \mapsto \mathtt{fcl}(\rho', f, x, M'), x \mapsto V'\} : \Gamma' + \{f : \tau_1 \to \tau_2, x : \tau_1\}$, by the induction hypothesis we obtain the relations:

$$\rho' + \{f \mapsto \mathtt{fcl}(\rho', f, x, M'), x \mapsto V'\} \vdash M' \Downarrow V \text{ and } \mathscr{R} \models V : \tau_2,$$

and we have V as the result of evaluation by the following evaluation derivation.

$$\frac{\begin{array}{c}\rho \vdash M \Downarrow \mathtt{fcl}(\rho', f, x, M') \quad \rho \vdash N \Downarrow V' \\ \rho' + \{f \mapsto \mathtt{fcl}(\rho', f, x, M'), x \mapsto V'\} \vdash M' \Downarrow V\end{array}}{\rho \vdash MN \Downarrow V}$$

Case *M is a finite map and* $\tau_1 = n$**.**
By the induction hypothesis, we have

$$\rho \vdash M \Downarrow \mathtt{fmap}(A_1^k \to V_1, \ldots, A_k^k \to V_k) \text{ and } \rho \vdash N \Downarrow A_i^k,$$

$$\mathscr{R} \models \mathtt{fmap}(A_1^k \to V_1, \ldots, A_k^k \to V_k) : n \to \tau_2 \text{ and } \mathscr{R} \models V_i : \tau_2 \text{ for all } i (1 \leqslant i \leqslant k).$$

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

By the definition of value typing, we have $k = k' = |\, n \,|$. Hence, we have an evaluation derivation:

$$\frac{\rho \vdash M \Downarrow \mathtt{fmap}(A_1^k \to V_1, \ldots, A_k^k \to V_k) \quad \rho \vdash N \Downarrow A_i^k}{\rho \vdash MN \Downarrow V_i}$$

and the result V_i has the relation $\mathscr{R} \models V_i : \tau_2$.

Case $M = \mathit{foreach}\ x \in M\ \mathit{with}\ [f,d]\ \mathit{do}\ N$**.**

The type is derived as follows:

$$\frac{\begin{array}{c}\Gamma \vdash M : \mu t.\tau(t) \\ \Gamma + \{f : n \to s, d : n \to \tau(n), x : \tau(n)\} \vdash M : \tau'(s) \\ n \text{ does not occur free in } \tau'(s)\end{array}}{\Gamma \vdash \mathit{foreach}\ x \in M\ \mathit{with}\ [f,d]\ \mathit{do}\ N : \mu t.\tau'(t)}$$

In the above, we can assume n and s are fresh node type variable and recursive type variable, respectively.

By the induction hypothesis, we have the following relations:

$$\begin{array}{l}\rho \vdash M \Downarrow X_i.\langle X_1 \to V_1', \ldots, X_k \to V_k'\rangle \text{ and} \\ \mathscr{R} \models X_i.\langle X_1 \to V_1', \ldots, X_k \to V_k'\rangle : \mu t.\tau(t).\end{array}$$

We can assume $|\, n \,| = k$ by proposition 2 (property 1). Let $Y_1, \ldots, Y_k$ be fresh node variables, $\mathscr{R}' = \mathscr{R} + \{s \mapsto \{Y_1, \ldots, Y_k\}\}$, and $S = [X_1 \mapsto A_1^k, \ldots, X_k \mapsto A_k^k]$. Then, by Lemmas 4 and 3, we have

$$\mathscr{R}' \models Y_i : s \text{ and } \mathscr{R}' \models S(V_i') : \tau(n) \text{ for all } i(1 \leqslant i \leqslant k).$$

Hence, under the recursive type variable environment $\mathscr{R}'$, the two finite maps have the following types:

$$\begin{array}{l}\mathscr{R}' \models \mathtt{fmap}(A_1^k \to Y_1, \ldots, A_k^k \to Y_k) : n \to s \text{ and} \\ \mathscr{R}' \models \mathtt{fmap}(A_1^k \to S(V_1'), \ldots, A_k^k \to S(V_k')) : n \to \tau(n).\end{array}$$

From these relations, we obtain

$$\begin{array}{l}\mathscr{R}' \models \rho_i : \Gamma + \{f : n \to s, d : n \to \tau(n), x : \tau(n)\} \text{ for all } i(1 \leqslant i \leqslant k), \\ \text{where } \rho_i = \rho + \left\{\begin{array}{l} f \mapsto \mathtt{fmap}(A_1^k \to Y_1, \ldots, A_k^k \to Y_k), \\ d \mapsto \mathtt{fmap}(A_1^k \to S(V_1'), \ldots, A_k^k \to S(V_k')), \\ x \mapsto S(V_i')\end{array}\right\}.\end{array}$$

By applying the induction hypothesis, we have the following relations:

$$\rho_i \vdash N \Downarrow V_i \text{ and } \mathscr{R} \models V_i : \tau'(s) \text{ for all } i(1 \leqslant i \leqslant k).$$

Hence, the expression is evaluated by the following derivation:

$$\frac{\rho \vdash M \Downarrow X_i.\langle X_1 \to V_1', \ldots, X_k \to V_k'\rangle \qquad \rho + \left\{\begin{array}{l} f \mapsto \mathtt{fmap}(A_1^k \to Y_1, \ldots, A_k^k \to Y_k), \\ d \mapsto \mathtt{fmap}(A_1^k \to S(V_1'), \ldots, A_k^k \to S(V_k')), \\ x \mapsto S(V_i')\end{array}\right\} \vdash N \Downarrow V_i \text{ for each } i, 1 \leqslant i \leqslant k}{\rho \vdash \mathit{foreach}\ x \in M\ \mathit{with}\ [f,d]\ \mathit{do}\ N \Downarrow Y_i.\langle Y_1 \to V_1, \ldots, Y_k \to V_k\rangle}$$

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

and the result value has type $\mathscr{R} \models Y_i.\langle Y_1 \to V_1, \ldots, Y_k \to V_k\rangle : \mu t.\tau'(t)$.

Case $M = up\ M\ with\ x_1 = N_1, \ldots, x_k = N_m\ end$.

The type is derived as follows:

$$\frac{\Gamma + \{x_1 : s, \ldots, x_m : s\} \vdash M : \tau(s) \quad \Gamma \vdash N_i : \mu t.\tau(t) \ \text{ for each } i,\ 1 \leqslant i \leqslant m}{\Gamma \vdash up\ M\ with\ x_1 = N_1, \ldots, x_m = N_m\ end : \mu t.\tau(t)}$$

By the induction hypothesis and Lemma 3, we have the following relations:

$$\begin{array}{l}\rho \vdash N_i \Downarrow X^i.\chi_i \text{ and } \mathscr{R} \models X^i.\chi_i : \mu t.\tau(t) \text{ for all } i(1 \leqslant i \leqslant m),\\ \rho + \{x_1 \mapsto X_1, \ldots, x_m \mapsto X_m\} \vdash M \Downarrow V \text{ and}\\ \mathscr{R} + \{s \mapsto \{X_1, \ldots, X_m\}\} \models V : \tau(s).\end{array}$$

Hence, the expression is evaluated by the following derivation:

$$\frac{\rho \vdash N_i \Downarrow X^i.\chi_i \ \text{ for each } i, 1 \leqslant i \leqslant m \quad \rho + \{x_1 \mapsto X_1, \ldots, x_m \mapsto X_m\} \vdash M \Downarrow V}{\rho \vdash up\ M\ with\ x_1 = N_1, \ldots, x_m = N_m\ end \Downarrow X.(\langle X \to V\rangle\ @\ (@_{i=1}^{m}\ \chi_i))}$$

Let $\chi_i = \langle X_1^i \to V_1^i, \ldots, X_{k(i)}^i \to V_{k(i)}^i\rangle$. We can assume by the renaming convention that the node variables X_j^i's are all distinct and $\{X_j^i \mid 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant k(i)\} \cap \mathscr{R}(u) = \emptyset$ for all $u \in Dom(\mathscr{R})$. By the definition of value typing, we have $\mathscr{R} + \{s \mapsto \{X_1^i, \ldots, X_{k(i)}^i\}\} \models V_j^i : \tau(s)$ for all $i(1 \leqslant i \leqslant m)$. Let $\mathscr{R}' = \mathscr{R} + \{s \mapsto \{X_j^i \mid 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant k(i)\} \cup \{X\}\}$. Then, by Lemma 3, we have $\mathscr{R}' \models V_i^j : \tau(s)$ for all $i, j(1 \leqslant i \leqslant m, 1 \leqslant j \leqslant k(i))$. Similarly, we also have $\mathscr{R}' \models V : \tau(s)$. Therefore, we can conclude that $\mathscr{R} \models X.(\langle X \to V\rangle\ @\ (@_{i=1}^{m}\ \chi_i)) : \mu t.\tau(t)$.

Case $M = dn(M)$.

The type is derived as follows:

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

By the induction hypothesis, we have the following relations:

$$\rho \vdash M \Downarrow X_i.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle \text{ and}$$

$$\mathscr{R} \models X_i.\langle X_1 \to V_1, \ldots, X_k \to V_k\rangle : \mu t.\tau(t).$$

Hence, the expression is evaluated by the following derivation:

$$\frac{\rho \vdash M \Downarrow X_i.\langle X_1 \to V_1, \cdots, X_k \to V_k\rangle}{\rho \vdash dn(M) \Downarrow S(V_i)}$$

where $S = [X_1 \mapsto X_1.\chi, \ldots, X_k \mapsto X_k.\chi]$ with $\chi = \langle X_1 \to V_1, \ldots, X_k \to V_k\rangle$. By the definition of value typing, we have $\mathscr{R} + \{t \mapsto \{X_1, \ldots, X_k\}\} \models V_i : \tau(t)$. Therefore, by Lemma 4, the result has type $\mathscr{R} \models S(V_i) : \tau(\mu t.\tau(t))$.

□

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

B Data-parallel suffix on binary tree

```
datatype α BTree = Leaf of α | Node of α * α BTree * α BTree;
datatype α State = Inactive | Active of α pBtree | Pebbled
     and α pBTree = pLeaf of α * α State
                  | pNode of α * α State * α BTree * α BTree;

suffix_tree opr T =
let fun init_tree T =
      foreach x in T with [f,d] do
        case x of Leaf(n) => pLeaf(n,Pebbled)
                | Node(n,t1,t2) => pNode(n,Inactive,f t1,f t2)
        end
      end
    fun strip pT =
      foreach x in pT with [f,d] do
        case x of pLeaf(n,_) => Leaf(n)
                | pNode(n,_,t1,t2) => Node(n,f t1,f t2)
        end
      end
    fun activate pT =
      foreach x in pT with [f,d] do
        case x of
          pLeaf(n,st) => pLeaf(n,st)
        | pNode(n,Inactive,t1,t2) =>
            let (n1,st1) = case (d t1) of
                             pLeaf(n,st) => (n,st)
                           | pNode(n,st,_,_) => (n,st)
                           end
            and (n2,st2) = case (d t2) of
                             pLeaf(n,st) => (n,st)
                           | pNode(n,st,_,_) => (n,st)
                           end
            in case (st1,st2) of
                 (Pebbled,_) => pNode(opr n n1,Active(f t2),f t1,f t2)
               | (_,Pebbled) => pNode(opr n n2,Active(f t1),f t1,f t2)
               | (_,_) => pNode(n,Inactive,f t1,f t2)
               end
            end
        | pNode(n,Active(t),t1,t2) => pNode(n,Active(f t),f t1,f t2)
        | pNode(n,Pebbled,t1,t2) => pNode(n,Pebbled,f t1,f t2)
        end
      end
    fun square pT =
      foreach x in pT with [f,d] do
        case x of
          pLeaf(n,st) => pLeaf(n,st)
        | pNode(n,Active(t),t1,t2) =>
            case (d t) of
              pNode(m,Active(t'),_,_) => pNode(opr n m,Active(f t'),f t1,f t2)
            | _ => (n,Active(f t),f t1,f t2)
            end
        | pNode(n,st,t1,t2) => pNode(n,st,f t1,f t2)
        end
      end
    fun pebble pT =
```

https://doi.org/10.1017/S0956796899003457 Published online by Cambridge University Press

```
      foreach x in pT with [f,d] do
        case x of
          pLeaf(n,st) => pLeaf(n,st)
        | pNode(n,Active(t),t1,t2) =>
            case (d t) of
              pNode(m,Pebbled,_,_) => pNode(opr n m,Pebbled,f t1,f t2)
            | _ => pNode(n,Active(f t),f t1,f t2)
            end
        | pNode(n,st,t1,t2) => pNode(n,st,f t1,f t2)
        end
      end
    fun pebble_game pT =
         case pT of
           ^pLeaf(_,Pebbled) => pT
         | ^pNode(_,Pebbled,_,_) => pT
         | _ => pebble_game (pebble (square (square (activate pT))))
         end
in
    strip (pebble_game (init_tree T))
end;
```

References

Bird, R. S. (1987) Introduction to the theory of lists. In: Broy, M., editor, *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag.

Blelloch, G. E. (1990) *Vector Models for Data-parallel Computing*. MIT Press.

Blelloch, G. E. (1993) *NESL: A nested data parallel language*. Technical Report CMU-CS-93-129, Carnegie Mellon University.

Bratvold, T. A. (1994) Parallelising a functional program using a list-homomorphism skeleton. In: Hong, Hoon, editor, *Pasco'94: First International Symposium on Parallel Symbolic Computation*, pp. 44–53. World Scientific.

Cole, M. (1989) *Algorithmic Skeletons: Structured management of parallel computing*. Pitman/MIT Press.

Damas, L. and Milner, R. (1982) Principal type schemes for functional programs. *Proc. ACM Symposium on the Principles of Programming Languages*, pp. 207–212.

Darlington, J., Field, A. J., Harrison, P. G., Kelly, P. H. J., Sharp, D. W. N., Wu, Q. and While, R. L. (1993) Parallel programming using skeleton functions. *Proc. PARLE93: parallel architectures and languages Europe. Lecture Notes in Computer Science 694*, pp. 146–160. Springer-Verlag.

Foisy, C. and Chailloux, E. (1995) Caml Flight: A portable SPMD extension of ML for distributed memory multiprocessors. In: Bohm, A. P. W. and Feo, J. T., editors, *High Performance Functional Computing*, pp. 83–96.

Forum, High Performance Fortran. (1993) *High Performance Fortran language specification.*

Gupta, R. (1992) SPMD execution of programs with pointer-based data structures on distributed-memory machines. *J. Parallel & Distributed Computing*, **16**, 92–107.

Hains, G. and Foisy, C. (1993) The data-parallel categorical abstract machine. *Proc. PARLE93: Parallel architectures and languages Europe. Lecture Notes in Computer Science 694.* Springer-Verlag.

Halstead, R. H. (1985) Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, **7**(4), 501–538.

Hammarlund, P. and Lisper, B. (1993) On the relation between functional and data par-

allel programming languages. *Sixth Conference on Functional Programming and Computer Architecture*, pp. 210–222.

Hatcher, P. J. and Quinn, M. J. (1991) *Data-parallel Programming on MIMD Computers.* MIT Press.

Hillis, W. D. and Steele Jr., G. L. (1986) Data parallel algorithms. *Communications of the ACM*, **29**(12), 1170–1183.

Hu, Z., Takeichi, M. and Chin, W.-N. (1998) Parallelization in calculational forms. *Proc. ACM Symposium on the Principles of Programming Languages*, pp. 316–328.

Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. and Perterson, J. (1992) Report on programming language Haskell a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, Haskell special issue, **27**(5).

Ito, T. and Matsu, M. (1988) A parallel lisp language PaiLisp and its kernel specification. In: Ito, T. and Halstead, R. H., editors, *Parallel Lisp: Languages and systems. Lecture Notes in Computer Science 441*, pp. 58–100. Springer-Verlag.

Jájá, J. (1992) *An Introduction to Parallel Algorithm.* Addison-Wesley.

Kahn, G. (1987) Natural semantics. *Proc. Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science 247*, pp. 22–39. Springer-Verlag.

Karp, A. (1987) Programming for parallelism. *IEEE Computer*, **20**(5), 43–57.

Keller, G. and Simons, M. (1996) A calculational approach to flattening nested data parallelism in functional languages. In: Jaffar, J. and Yap, R. H. C. (eds), *Concurrency and Parallelism, Programming, Networking, and Security: Second Asian Computing Science Conference, ASIAN'96. Lecture Notes in Computer Science 1179*, pp. 234–243. Springer-Verlag.

Lasser, C. (1986) *The Essential *Lisp ¡anual.* Thinking Machine Corporation, Cambridge, MA.

Leroy, X. (1992) *Polymorphic typing of an algorithmic language.* PhD thesis RR-1778, INRIA, France.

Loulergue, F. and Hains, G. (1997) Functional parallel programming with explicit processes: beyond SPMD. In: Lengauer, C., Griebl, M. and Gorlatch, S., editors, *Euro-PAR'97 Parallel Processing. Lecture Notes in Computer Science 1300*, pp. 530–537. Springer-Verlag.

Loulergue, F., Hains, G. and Foisy, C. (1998) A calculus of recursive-parallel BSP programs. *CMPP'98 International Workshop on Constructive Methods for Parallel Programming.* Marstrand, Sweden. (Technical Report, University of Passau, Germany.)

Merrall, S. and Padget, J. (1992) Plural EuLisp: A primitive symbolic data parallel model. *Lisp and Symbolic Computation*, **6**(1/2), 201–219.

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML.* MIT Press.

Misra, J. (1994) *Powerlist: A structure for parallel recursion.* Prentice Hall.

Nishimura, S., Ohori, A. and Tajima, K. (1996) An equational object-oriented data model and its data-parallel query language. *Proc. OOPSLA 96*, pp. 1–17.

O'Donnell, J. T. (1993) Data parallel implementation of extensible sparse functional arrays. *Proc. PARLE93: Parallel architectures and languages Europe. Lecture Notes in Computer Science*, pp. 68–79. Springer-Verlag.

Prins, J. and Palmer, D. (1993) Transforming high-level data-parallel programs into vector operations. *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 119–128.

Rogers, A., Carlisle, M. C., Reppy, J. H. and Hendren, L. J. (1995) Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, **17**(2), 233–263.

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

Sabot, G. (1988) *The Paralation Model: Architecture-independent parallel programming*. MIT Press.

Skillicorn, D. B. (1994) *Foundations of Parallel Programming*. Cambridge Series in Parallel Computation 6. Cambridge University Press.

Suciu, D. and Tannen, T. (1994) Efficient compilation of high-level data parallel algorithm. *Proc. ACM Symposium on Parallel Algorithms and Architectures*.

Talpin, J.-P. and Jouvelot, P. (1993) Compiling FX on the CM-2. *Static Analysis (WSA '93). Lecture Notes in Computer Science 724*, pp. 87–98. Springer-Verlag.

Tofte, M. (1988) *Operational semantics and polymorphic type inference*. PhD thesis CST-52-88, Department of Computer Science, Edinburgh University.

Wholey, S. and Steele Jr., G. L. (1987) Connection machine lisp: A dialect of common lisp for data parallel programming. *Proc. International Conference on Supercomputing*.

Wyllie, J. C. (1979) *The complexity of parallel computations*. Technical Report TR-79-387, Department of Computer Science, Cornell University, Ithaca, NY.

Yuasa, T. (1992) A SIMD environment TUPLE for parallel list processing. *Parallel Symbolic Computing: Languages, systems, and applications. Lecture Notes in Computer Science 748*, pp. 268–286. Springer-Verlag.