International Journal of Parallel Programming, Vol. 29, No. 4, 2001

Enhancing Functional and Irregular Parallelism: Stateful Functions and their Semantics

Isabelle Attali,[1] Denis Caromel,[1] Yung-Syau Chen,[2] Jean-Luc Gaudiot,[2,4] and Andrew L. Wendelborn[3]

Received November 1999; revised July 2000

We describe an approach in which stateful computations can be expressed within the framework of a functional language. We consider algorithms with nondeterministic intermediate results and a deterministic final result which is obtained for any series of intermediate values of some variable shared among parallel tasks or, in other words, the ordering of updates to the variable does not matter. Functional languages normally abstract away from explicit synchronization and exploit parallelism between separate uses of a variable. But in some cases we can relax that requirement with both parallelism and determinate computation. To increase its expressiveness and efficiency for this important class of problems, we propose to extend the Sisal language with state variables encapsulated within stateful functions. We have used Centaur to specify and construct a semantic-based environment. We illustrate the proposed language extension with analysis of several examples, and comparison with other languages.

KEY WORDS: Irregular parallelism; single assignment languages; data flow; Sisal; natural semantics.

[1] INRIA Sophia Antipolis, Université de Nice, Sophia Antipolis, BP 93, 06902-Sophia Antipolis Cedex, France.

[2] Department of Electrical Engineering, 3740 McClintock Avenue, EEB336, University of Southern California, Los Angeles, California 90089-2563.

[3] Department of Computer Science, University of Adelaide, Adelaide, SA 5005, Australia.

[4] To whom correspondence should be addressed. E-mail: gaudiot@usc.edu

0885-7458/01/0800-0433$19.50/0 © 2001 Plenum Publishing Corporation

1. INTRODUCTION

Due to the high cost of complex data-flow architectures, the execution model for exploiting fine-grain parallelism has evolved gradually from the data-driven to the multiprocess execution model.[1] The advantage of functional languages is the ease of parallelism extraction.[2] Functional languages can be used to write programs for multiprocess systems because they can expose a high degree of parallelism. The processing power of multiprocess architectures is useful only when the programming language can effectively take advantage of it.

We consider, for a moment, some characteristics of "*imperative*" and "*functional*" programming languages.

Since imperative languages derive from a sequential machine model, they are not easily compiled for parallel execution.[3] In the general case, any variable in an imperative language program is updatable over the whole program. An imperative language program cannot be executed in parallel on a multiprocessing system without using complex synchronization mechanisms to avoid, for example, access conflicts to shared variables.

For example, in the following imperative program segment, a single variable a is used, in two distinct parts of the program, and is updated at each use.

```
/* chunk_1: */
   a = 1; b = 2; c = 3; delta = b ^ 2 - 4*a*c;
...
/* chunk_2: */
         a = 5; y = f(a); ...
```

Assume that the first chunk of commands is scheduled in processor P, and the second chunk of commands is scheduled in processor Q. These two code sequences cannot be executed in parallel without synchronization between P and Q, otherwise arbitrary interleaving of the code sequences is possible, raising the possibility that either P or Q may use an incorrect value of a: P may see the value of a as 5, and Q may see it as 1. In fact, the variable a as used in chunk_1 is logically independent of the variable a in chunk_2: the same storage location is used for both, necessitating special coordination code for use of that location by parallel processes.

On the other hand, in a functional language, single-definition rules ensure that this situation is avoided, and thus the processes in P and Q can be executed in parallel without interfering with each other.

Sisal and HASKELL are examples of general purpose functional languages without program constructs which would induce side-effects.[4] As indicated earlier, this property can be exploited to yield quite straightforward algorithms for detection of parallelism in functional programs, and

generation of parallel code. In many cases, such techniques can be utilized very effectively to produce high performance programs.[5]

Nonetheless, there are some classes of program where purely functional programming can hinder rather than enhance expression and analysis of parallelism. We examine some such programs in this paper, in particular, for algorithms (for example, shortest path) where it is natural to define a single shared variable to record the best solution (such as the length of the shortest path) found so far, but where we can allow parallel processes to update that variable in any order; provided that the update operation is itself atomic, it is not necessary for any further coordination of accesses. Functional languages normally abstract away from explicit synchronization by insisting that each separate use of a variable be separately identified, and then exploiting parallelism between those separate uses. But in some cases we can relax that requirement and still have parallelism and determinate computation of the overall program result.

We discuss an extension of Sisal with a form of *stateful computation*, so that both expression and performance of such applications can be improved.[6, 7] From the specifications of the syntax and semantics of the language,[8, 9] a development and visualization environment for extended-Sisal programming is produced.

This paper demonstrates how a functional language can be extended with stateful computations to improve both the programmability and the performance of real world applications, while keeping the ease of parallel execution that comes naturally with functional languages.

Section 2 reviews related work. Section 3 offers the motivation for our work. Our methodology and the extended features are described and discussed in Section 4. The implementation of extended Sisal is described in Section 5: syntax and semantic specifications are shown and discussed. Programs for some parallel applications and the performance evaluation are presented in Section 6. Finally, conclusions are presented in Section 7.

2. RELATED WORK

Several attempts have been made to include stateful computations in a functional language and still maintain ease of extraction of parallelism.[10–12] The languages Scheme[13] and ML[14] each support both functional and imperative programming; we do not consider them in detail here, as our emphasis is on approaches that can be considered as occasional departures from a primarily functional base. In this section, some earlier approaches to stateful computations in functional programming languages are discussed.

2.1. Id

ID[15] designers provide a series of abstractions for the declarative expression of different paradigms of programming, beginning with a functional core, then *I*-structures[16] as write-once structures with determinate nonstrict evaluation, accumulators[17] to provide a style of programming similar to logic programming, whereby a variable reaches its final value incrementally, through separately specified refinements. Finally, *M*-structures[18] provide high-level specification of updatable structures, allowing multiple assignments.

Solutions based on accumulators are quite concise and readable, and are potentially amenable to implementation as updates without unnecessary sequentialization. However, central to the notion of accumulator is the definition of its result in terms of a list comprehension; the sequentialization implied by this comprehension can make it difficult to realize such an implementation, and the necessity to specify accumulation by a binary operator is somewhat restrictive: greater generality is provided by both *M*-structures, and the stateful functions introduced in this paper.

M-structures are intended to provide a high-level, yet efficient, description of updateable structures. *M*-structure primitives rely on implicit synchronization to provide atomicity of operations, and strictness of evaluation context to ensure that such operations are ordered correctly, and that reads and writes are balanced: one read must be matched by exactly one write. As ID assumes nonstrict, eager, parallel semantics by default, a barrier construction is introduced into the language to provide explicit sequencing where necessary. The state variables introduced later are in some ways similar to *M*-structures, but we believe that programming with state variables is less complex because we can statically determine synchronization requirements, and automatically generate the necessary code, rather than require explicit specification of synchronization.

ID programming with *M*-structures is based on the notation of comprehensions. It is our goal in this paper to provide a similar convenient, appropriately constrained abstraction of state variables, but in an idiom natural to Sisal.

2.2. Monads

The concept of monads is an important tool for functional programmers because it provides a framework for describing a wide range of programming features, including I/O and state.[4, 10, 19, 20]

Wadler[20] shows that, under certain conditions, operations on a monad can be described using the notation of comprehensions. The comprehension provides a mechanism for selecting and operating on elements of the monad. The commonly-used list comprehension is a concise and powerful mechanism for generating and filtering lists; analogously, in the more general case of a monad, a comprehension provides a layer of

abstraction beneath which more complex and interesting behaviors can be encapsulated. It also provides an ordering of operations on the monad.

A monad can be defined to encapsulate state. Monad laws ensure (as indicated earlier) that the state is handled in an appropriate way.[20] Hence, programs using monads can be reasoned about using referential transparency, in the usual way as in a pure functional language.

Monads simplify the programming of state manipulation in a functional manner. Consider the use of a counter to generate unique names. An impure program (in C, for example) can simply update a variable. A purely functional program must represent the state explicitly: it can mimic a counter by passing the value of the counter into every function which uses or changes it, and returning as a result the new value of the counter. This has the advantage that all uses of state in a program are exposed, but the "plumbing" of all state components through all functions that use them can be very tedious. Monads preserve the functional semantics of state manipulation, but hide the details of plumbing under a higher level abstraction.

Clearly, in this name-generation application the programmer needs a global counter and that is implemented as an implicit state.

Although a monadic program is an elegant and powerful way of using state (the counter), while maintaining referential transparency, it has the following limitations. First, it is difficult for the implicit state method to express more than one state component (e.g., two counters). Secondly, the implicit state method is an abstract form of passing around parameters, and thus inevitably imposes an artificial serialization in the program. Note that such serialization is not always inherent in the applications. Indeed, no order of new names is predetermined in the algorithm.

In this paper, we propose a representation of state manipulation, augmenting a strict functional language, that imposes minimal serialization of operations, and allows expression of a high level of parallelism in applications over irregular structures in which ordering of operations does not matter in producing the final result. Our representation differs from the monad-based approach in that it admits non-functional constructions, but in a manner that clearly identifies stateful aspects, and can be used and understood orthogonally to the functional core of the language.

3. MOTIVATION

Access to high-level abstractions and implicit parallelism allow functional language programmers to concentrate on the implementation of the applications without being concerned with low-level execution details. However, for applications with stateful computations, the expressive power

of functional languages is frequently insufficient. Functional languages remove some of the burden of managing parallelism, but several types of computations are difficult to express without state. We found that many problems caused by such limitations of the functional programming can be ameliorated by allowing a form of update on a small number of variables. We thus introduce an extension to the functional programming paradigm.

In Section 2, we considered the example of a counter generating unique names, and observed that, in both a purely functional program and its more concise monad-based equivalent, artificial serialization is imposed, and extraction and exploitation of parallelism thus made more difficult.

We now consider another example, in which the efficient parallel execution of the branch-and-bound application (Fig. 1) is facilitated by using a single variable, visible to all participating processes. In this application, we try to find the shortest path from starting node A to goal node J.

Suppose that initially there are three parallel execution processes. The first execution process performs the calculation for the path: $A \rightarrow B \rightarrow E \rightarrow J$. The second execution process performs the calculation for the path: $A \rightarrow C \rightarrow \cdots$, and the third execution process performs the calculation for the path: $A \rightarrow D \rightarrow \cdots$

Assume the following scenario: when the first execution process terminates, the value of a global variable M, representing the minimum so far, is assigned 5. Then the second process performs the calculation until it reaches node F. The accumulated path length is 7, which already exceeds the value of M. The whole subtree rooted at F can thus be pruned. When J is reached through G, the calculated result is 4, which is less than the current value of M, so M is reassigned 4.

This is an example of an algorithm with nondeterministic intermediate results and a deterministic final result, in that the same final result is

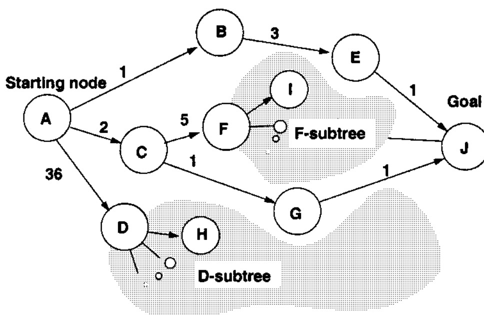

Fig. 1. Branch-and-bound application.

obtained for any series of intermediate values of M: in other words, the ordering of updates to M does not matter.

Thus, we see that the use of a shared variable, greatly facilitates ease of programming and reduction of wasteful computation for this application. Such use of a shared variable cannot be programmed directly in a pure single-assignment language. Further, such a scheme cannot be achieved by directly applying monads, I-structure or M-structures in the programs.

4. THE STATEFUL EXTENSION

4.1. Principles

Here, we describe an extension to the language Sisal that will allow ease of programming and efficient implementation of algorithms like those discussed previously. In so doing, our goal is to maintain consistency with Sisal idioms, and to not compromise its primary goal of determinate parallel programming.

While adding state to a single assignment language such as Sisal, one has to respect its philosophy. We introduce declaration of state through an extension of the notion of function, and update of state in a manner similar to the notion of re-binding already present in the language.

First, we summarize Sisal scope rules. Sisal does not permit any data values to be imported from any enclosing function definition; in other words, data is presented to a function by the explicit data flow of parameter passing. Function names themselves are inherited by all nested function definitions, while redefinition of function names inherited from an outer scope is not permitted. We want to change that as little as possible.

The stateful extension is introduced by mean of *state variables*, which are declared with the keyword state:

```
function h (returns integer,...)
  state
        s1, s2 : integer
              ...
      end state;
            ...
end function
```

We call any function declaring a state variable a *stateful function*. An intuitive description of the syntax and semantics of state variables and stateful functions follows, with more detail in the next section.

We adopt two important principles:

1. a stateful function can be written without any restriction (any legal Sisal definition is legal in a stateful function), and
2. state variables retain their values between each function invocation.

The first principle is motivated by our desire to write definitions within stateful functions in the same way as we write definitions in conventional Sisal functions. The second expresses the state-oriented behaviour of these functions.

State variables are accessible only via a stateful function. The stateful function defines, for each of its state variables, both an initial value, and a rule for updating the state variable. An update is expressed as a re-binding of the variable to a new value, using the same notation as for re-binding a Sisal loop variable.

Figure 2 gives a few examples of stateful functions with state variables. They comply with static Sisal scoping rules. Sharing of a state variable is effected by having its definition only in the stateful function where it is declared; it is not a global variable. This is consistent with the earlier analysis of the shortest path algorithm, where the requirement is not that the variable be global, but that it be shareable among the processes participating in the computation of the minimum; the mechanism used confines visibility of state variables to those functions that need them. Stateful functions require special treatment with respect to parallel execution. This will be further detailed in the next section.

```
function main (returns integer, ... )

   state s0 : integer initial s0 := 1 end state;

   function h(returns integer, ... )
      state s1,s2 : integer
         initial
           s1 := 0 ;
           s2 := g (0.0)
      end state;

      function g ( p : real returns integer)
         ...  % p only accessible
      end function

      ... % s1, s2 can be used here, and have to be defined
      ... % s0 is not accessible within h
      ... % (Sisal scope rules)
      s1 := if ... then ... else ... end if
      s2 := g (old s2)
      ...
   end function

   ... % s0 can be used here
   ... % s1 and s2 cannot be used here -- scope within h only
   ... %    i.e. s1, s2 updated only by invoking h

   ... % the result returned from h will
   ... % usually be expressed in terms
   ... % of the state variables

end function % main
```

Fig. 2. State variables example.

4.2. Parallel Executions

In this section we specify in more detail the parallel semantics of stateful functions and indicate how they can be exploited in a parallel execution.

We use as guidance in defining the semantics of state variables, the existing semantics of Sisal loops. We can see in Fig. 3 a very simple Sisal loop in which a loop variable *n* is first initialized, then rebound to an updated value at each iteration (using its *old* value at the previous iteration).

In the proposed extension, the function body specifies a series of one or more updates to each of its state variables. Each such update takes the form of a re-definition of the state variable, in a manner similar to the re-definition of a loop variable. One such definition specifies a transition from one value of the state variable to the next. The function body can specify several such transitions, as a series of updates to the state variables, that can be executed in parallel. Later, we discuss the precise mechanism. The body of the stateful function can be either a single definition, specifying an updated value of the state variable, or a loop, used as the mechanism for specifying a series of state transitions. The body of the loop specifies a single update of each state variable.

Each updating definition is of the form (for state variable v)

```
v := Exp;
```

where Exp represents an arbitrary expression defining the computation of the updated value.

In the expression that defines the updated value, the previous value is bound to old v, and the actual variable v itself is undefined. Outside that expression, v refers to the updated value, and old v to the previous value. Here, each invocation of a stateful function corresponds to a new binding of the state variable.

The initial value is specified separately, within the variable declaration with the keyword initial:

```
for
    n := 0
while n < 5 do
  n := old n + 1
  returns n
end for
```

Fig. 3. A simple Sisal loop.

```
function h (returns integer,...)
  state
        s1, s2 : integer
          initial
            s1 := ...;
            s2 := ...
      end state;
      ...
end function
```

The semantics being that initial statements are executed exactly once upon the first invocation of a stateful function. The variables introduced by the state construct have the scope of the current function. Within the *initial* construct, the visible names are, as defined by standard Sisal rules, visible function names and formal parameters of the current function. Of course, stateful variables cannot be used in the right-hand side part of the construct. One and exactly one initial value must be given to a state variable.

We now look in more detail at some aspects of the semantics. Our model of a stateful function assumes that it defines an update, or transition, from one state to another. However, if the update merely preserves the value (such as v := old v), its specification may be omitted.

The expression component of an update rule, from the point-of-view of translation to parallel execution, is regarded as atomic. An implementation can then treat each individual update as a critical section, and schedule them appropriately. If the hardware has a concurrent coordination primitive such as fetch-and-add,[21] then these updates can be executed in parallel, thereby providing a highly efficient implementation of a series of updates in which the order does not matter.

We stipulate, in the interest of keeping update code simple, that the update rule should not include any call to another stateful function, nor any (mutual) recursion onto the stateful function itself. Reading the value of a state variable is, of course, assumed to be atomic. The update rule of a state variable can include the use of other state variables, namely those declared within the same stateful function.

For an array declared as a stateful variable (a stateful array), whatever the dimension of the array, each element is considered to be a stateful variable by itself. In other world, the granularity of mutual exclusion for a stateful array is each cell of the array. Such a structure can be implemented by using a mechanism similar to *I*-Structures[16] or a memory architecture with element level synchronization as in the Tera supercomputer.[22]

Figure 4 presents a stateful function returning and storing a maximum value. In this example, stateful function Gmax encloses a special variable max, initialized to argument v. The body of Gmax incorporates the update

```
function Gmax (v: integer returns integer)
   state max: integer
      initial
         max := v
   end state;

   let
      max := if v > old max then v else old max
             end if
   in
      max
   end let
end function
```

Fig. 4. A stateful function collecting a maximum value.

rule for max: if the argument v is larger, max is updated to that value, otherwise it does not change. In the context of, for example, a branch-and-bound algorithm (see later), Gmax may be invoked many times by the parallel tasks of the computation. This mechanism ensures efficient realization of these unserialized update operations.

5. SPECIFYING STATEFUL FUNCTIONS

We have extended the Sisal language with user-declared state variables. A state variable is to be treated as a state variable within the scope of the function in which it has been declared. One of the benefits of functional languages—easy parallel processing—is kept, because only the state variables are of concern in terms of nondeterminacy; this can be easily singled out and taken care of. This extension of the Sisal language has been formally described using previous work[8, 9] on the formal definition (both syntax and dynamic semantics) of the Sisal language[5] using the CENTAUR system.[23] In the following, we first detail how we extended the syntactic constructs in order to introduce state variables, and then how this influences the formal semantics of the language.

5.1. Scope and Goal of the Specification

As in our previous work, we assume that the Sisal program is correct with respect to static semantics (such as enforcement of the scoping rules and type-checking), and we consider here only dynamic semantics. For instance, it is statically checked in the program that a state variable cannot be used outside its scope (the scope of its function definition). Static semantics also addresses the check that updates of stateful variables do not include calls to stateful functions, nor any direct or indirect recursion in the current stateful function. Another role of static semantics is to provide unique names for state variables. A variable is named according to the nesting of the function in which it is defined. For example, a state variable v, defined in a function f, itself defined in g, will be named $g\#f\#v$ after static checking, ready for interpretation using dynamic semantics.

A formal definition permits to fully express our extension without omission or ambiguity, without contradicting the existing language definition for Sisal. The Sisal reference manual[5] defines (informally) the language but does not define its parallel execution; our semantics defines (formally) the stateful Sisal extension but does not provide a parallel execution model. Indeed, our formal semantics captures the meaning of a Sisal program (i.e., the results it produces). The strategy that we are actually using in the semantics (for both Sisal and stateful Sisal), is a sequential eager left-to-right evaluation. Formal semantics does not capture the parallelism that can be extracted from the program (it is independent from any parallel evaluation strategy). To specify the parallelism that can be extracted, one needs a specific model (e.g., IF1[24] and IF2[25]), which then could be used to specify parallelism with respect to state variables. IF2 is a refinement of IF1, incorporating a model of storage used primarily for expressing trade-offs between storage and parallelism. However, such a precise and formal definition of a parallel evaluation strategy is outside the scope of the current paper. One difference between Sisal and Extended Sisal regarding nondeterminism is that, whatever the evaluation strategy is, the result of a Sisal program is always the same. On the other hand, one can write a stateful Sisal program that produces nondeterministic results; in that case our semantics just reflects one of them. The intent is that programmers use the stateful functions to write programs that still have a deterministic result. Of course, this cannot be checked.

Finally, our approach—design and definition—was successful in the sense that the formal definition had an impact on the design of extension itself (on the scoping rules of state variables for instance); the formalization also validates the extension within the scope of our existing formal semantics. We believe that all this makes our stateful extension elegant and simple.

Within the CENTAUR system, METAL is a meta-language used to specify the syntax of programming languages. Writing a METAL specification for a given language is the first step towards specifying this language and constructing its environment. TYPOL is a programming language that implements Natural Semantics.[26] Semantic specifications are operational, using a logical style (inference rules over typed sequents), both high-level and executable. We describe and comment the syntactic and semantic specifications in the following sections.

5.2. Syntactic Specifications

In METAL, one can specify the concrete syntax of the language, an abstract syntax, and building rules from the former to the latter (how to build an abstract syntax tree during parsing).

A programmer with stateful functions can insert declarations of state variables between a function header and the associated function body. This is expressed in the following extended definition of a function definition:

```
< Function_Def > ::= "function"  < Function_Header >
                                 < Opt_State_Variable_List >
                                 < Expression_List >
                        "end" "function";
    function_def( < Function_Header > ,
                  < Opt_State_variable_List > ,
                  < Expression_List > )
```

The < Opt_State_variable_List > nonterminal needs to be further specified in the METAL specification. The first part of the definition expresses concrete syntax, including keywords. The second part describes how to build an abstract syntax tree for function definition. This operator function_def is now extended with a (possibly empty) new subtree (denoting the optional list of declared state variables) as follows:

```
function_def -> Function_header State_variables Expressions;
```

Some more METAL rules describe the construction of the State_variables subtree in the case of the declaration of zero, one or more state variables; each state variable has a type and one and exactly one initial value, the result of the evaluation of the corresponding expression. Initialization expressions are usual Sisal expressions except only constants and parameters can be used as leaves.

The associated abstract syntax for State_variables is the following:

```
State_variables -> State_variable *;
State_variable -> Identifier Type Expression;
```

A list of state variables is made of triples of identifier, type, and expression. Type is the declared type for the state variable, and Expression is the expression denoting its initial value.

The design of extended Sisal is intended to make the task of programming easier by allowing higher expressiveness. More convenience for programmers conversely means more challenges in semantic specifications. We will show semantic specification for state variables in the following section.

5.3. Semantic Specifications

We do not detail here the semantic definition of Sisal (the reader can refer to Attali *et al.*[8, 9]). We just recall the principles: specifications are

organized in modules, each of which deals with similar concerns. The starting point of the semantics is to evaluate the main function body given its name and its arguments. The result of the semantics is then a list of values.

Values in Sisal are either constants (boolean, integer, real, complex, double, character, string, errors), or arrays, union, records, streams, or closures, which are pairs of λ-expression (representing the function body) and environment.

Environments are necessary to manage the binding between identifiers and values since the value of an expression depends on the values of identifiers that occur free in it, including bindings for function names, when they are declared via an assignment in a *let* or a *for* construct. We define an environment as a list of pairs composed from a name and a value. Because standard Sisal is a purely functional language, this environment appears in most predicates as a parameter but is not modified during expression evaluation. This is formalized in the predicates by the fact that the environment is not returned as a result. Introduction of state variables requires to build a specific environment for these variables: we build STATE_ENV as a list of pairs made of Identifier and Expression_or_Value. This is almost the abstract syntax for State_Variables except that we omit the type of state variables.

In the state variables environment, identifiers of state variables are first associated with the initialization expression. On the first invocation of the corresponding stateful function, this expression is evaluated into a value which then updates the expression in the environment. This is why the second element in the pair can be either an expression or a value Type Expression_or_Value.

Moreover, as state variables will be updated throughout execution, the STATE_ENV environment also appears in predicates as a result.

The semantics of an extended Sisal program is expressed in the following starting inference rule, updated for our extension as follows:

```
main_rule:
function_definition(FNAME |- PROG : true(),
                   function_def(function_header(_,FOR_PARAMS,_),
                                STATE_ENV,
                                EXPS)) &
bind_parameters(FOR_PARAMS,
EFF_PARAMS -> BIND_PARAMS) &
function_execution(SYSTEM, BIND_PARAMS, STATE_ENV
                   |- EXPS: VALUES, STATE_ENV')
-----------------------------
FNAME, EFF_PARAMS |- PROG : VALUES;
```

Given a name and actual parameters, the result of the semantics is a list of values, provided that the function is actually defined in the current program. The binding between formal and actual parameters is accomplished in the *bind_parameters* predicate. The function is then executed (*function_execution* predicate) in the environment reduced to this binding (BIND_PARAMS), and the state variables environment (STATE_ENV). Execution of the function may update state variables (which then results in STATE_ENV′).

The following rules show how the relationship between the calling function and the called function is handled with regard to state variable environments.

```
first_invocation:
function_definition( NAME |-  PROG : true( ),
                    function_def( function_header( NAME,
                                  FOR_PARAMS, T ),
                                  STATE_ENV',
                                  EXPS ) ) &
eval_expression_list( PROG, ENV, STATE_ENV
                    |-  EXP_LIST -> VALUES, STATE_ENV1 ) &
bind_parameters( FOR_PARAMS, VALUES -> BIND_PARAMS ) &
init_state_variables( STATE_ENV', BIND_PARAMS -> STATE_ENV'' ) &
appendtree( STATE_ENV1, STATE_ENV'' -> STATE_ENV2 ) &
function_execution( PROG, BIND_PARAMS, STATE_ENV2
                    |-  EXPS : VALUES', STATE_ENV3 )
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
SYSTEM, ENV, STATE_ENV
          |- invocation( NAME, EXP_LIST ) : VALUES', STATE_ENV3;
          provided stateful( NAME ) & first_call( NAME );

other_invocations:
function_definition( NAME |-  PROG : true( ),
                    function_def( function_header( NAME,
                                  FOR_PARAMS, T ),
                                  _,
                                  EXPS ) ) &
eval_expression_list( PROG, ENV, STATE_ENV
                    |-  EXP_LIST -> VALUES, STATE_ENV1 ) &
bind_parameters( FOR_PARAMS, VALUES -> BIND_PARAMS ) &
function_execution( PROG, BIND_PARAMS, STATE_ENV1
                    |-  EXPS : VALUES', STATE_ENV3 )
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
SYSTEM, ENV, STATE_ENV
        |- invocation(NAME, EXP_LIST) : VALUES', STATE_ENV3;
        provided stateful(NAME) & not_first_call(NAME);
```

These two inference rules for function invocation of stateful functions can be explained as follows:

1. two cases occur depending if it is the first invocation of the function or not (*first_call* predicate);
   - on the first invocation, initialization expressions are evaluated into values and used to make a specific state variable environment;
   - on following calls, the state variables declared in the stateful function are already stored in the state variable environment; we do not need to retrieve the corresponding subtree in the function definition (use of the anonymous variable "_");
2. the function has been defined as a stand-alone unit: the *function_definition* predicate returns *true* and the actual definition for function of name NAME;
3. the environment (for standard variables) in which the function body will be evaluated only comprises the binding between its formal and actual parameters (BIND_PARAMS, result of the call to the *bind_parameters* predicate);
4. evaluation of effective parameters (*eval_expression_list* predicate) may result in an update of the current state variable (STATE_ENV1);
5. the computation of the current state variables environment for the function execution differs in the two rules:
   - on the first invocation, initial values of state variables are combined with the current state variable environment (*appendtree* predicate);
   - on following calls, the current state variables environment is directly used;
6. execution of function NAME is handled in the two following environments: BIND_PARAMS and the current state variables environment (STATE_ENV2 or STATE_ENV1, depending on first invocation or not). This execution (*function_execution* predicate) results into a list of values VALUES' and updates the state variable environment into STATE_ENV3.

The *let* and *for* constructs are used to assign variables (either conventional or state). The list of assignments is then treated in the semantics on the basis of the corresponding syntactic construct (list of assignments), which means that each assignment is examined in sequence (left-to-right), with the use of unification for state variables.

This is to respect the treatment of conventional variables, as well as to handle the update of state variables, in which, as mentioned in Section 4.2, the order is not important. We reflect this in the semantics with the use of unification during evaluation of right-hand sides of assignments: in the assignment v := Exp, Exp has to be evaluated in the context of update of the state variable v. During evaluation of Exp, the actual variable v itself is undefined, it is in fact a "free" logical variable, bound to its future value, the Exp to be evaluated. Its previous value can be referenced with old v.

For every other assignment in the sequence, depending on the actual order of evaluation, either v is already updated, or v remains to be updated with the result arising from the symbolic evaluation of the corresponding expression.

The following rules show the formalization of the assignment of variables (coming from a *let* or a *for*). Two cases occur, depending on the status of the variable. The variable is a conventional variable and a new pair made of a name and a value is added to the environment (the state variables environment remains unchanged, rule *assign_variable*). On the other hand, in the case of a state variable, the usual environment is unchanged and the value of the state variable is updated in the state variables environment (rule *update_state_variable*).

```
assign_variable:
ENV, STATE_ENV |- NAME, VALUE -> env[pair(NAME,
                VALUE).ENV], STATE_ENV;
              provided not_stateful(NAME);

update_state_variable:
update(STATE_ENV |- NAME, VALUE -> STATE_ENV')
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ENV, STATE_ENV |- NAME, VALUE -> ENV, STATE_ENV';
              provided stateful(NAME);
```

In summary, the state variable extension required minimal changes in the formal semantics. These changes are the following:

- add state variable environments (parameter and result) in every rule dealing with expressions (purely syntactical change);

- add new rules for function invocation to handle the case of stateful functions and taking care of their first invocation (rules *first_invocation* and *other_invocations*);
- add a new rule dealing with assignment of state variables (updating the state variable environment instead of augmenting the usual environment, rule *update_state_variable*);

6. COMPARING SISAL AND EXTENDED SISAL

The pure Sisal language and the extended Sisal we propose are compared here with regards to programmability, parallelism, and performance.

6.1. Newnames

The newnames application generates different names, and each name can be used in different functions. The Sisal program must be implemented by passing around parameters as shown in Fig. 5. The execution results are as follows: *values[101, 102, 103, 104, 105, 106, 107, 108, 109, 110]*.

In this program, we have explicitly enumerated all the names to be used, rather than generating them with, for example, an iterative construct. The main reason for that is to facilitate maximal parallelism in the generation of these names: in the Sisal program of Fig. 6, there are ten pairs of definitions; the two definitions of each pair are independent and specify potential parallelism, but parallelism across the ten pairs is restricted by the sequentialization arising from data dependencies a1 on a0, a2 on a1, and so on. We have shown here that a program with such artificial sequentialization can be made parallel using stateful functions, and that this parallelism can be efficiently implemented as concurrent increment operations on a shared variable representing the name. With that done, there is parallelism both within and across the definition pairs.

It is possible to specify the generation of names with a loop, rather than using a lengthy enumeration, but it turns out that potential

```
function sn(n: integer returns integer)
        100+n
end function
function nn(m: integer return integer)
        m + 1
end function
function main(returns integer, integer, ..., integer)
 let a0 := 0; a1 := nn(a0);
     n1 := sn(a1); a2 := nn(a1);
     n2 := sn(a2); a3 := nn(a2);
        .
     n10 := sn(a10);
 in n1, n2, ... , n10
 end let
end function
```

Fig. 5. Newname program in Sisal.

```
function main(returns integer, integer, ..., integer)
   function gn(returns integer)
      100+ f(1)
   end function

   gn(), gn(), ..., gn()
end function

function f(v: integer returns integer)
   state x: integer
      initial
         x := 0
   end state;

   let
      x := old x + v
   in
      x
   end let
end function
```

Fig. 6. Newname program in extended Sisal.

parallelism is then limited. Take the following program fragment as an example:

```
for n in 1, 10
   do
      p := nn(i);
      q := sn(p);
   returns array of q
end for
```

This is a parallel loop, and its range generator does indeed specify ten independent instantiations of the loop body, with the result of each instantiation (a name q) gathered into the corresponding position of the result array. Now, construction of the array is not sequentialized by this ordering, and thus can take place in parallel. The restricted parallelism occurs as a result of strictness of evaluation applied to the result array; no element is available for use elsewhere in the program until the array value is completely constructed. Although an implementation of Sisal with nonstrict arrays (for example, arrays implemented as *I*-structures) can be envisaged, it is more usual to implement Sisal arrays strictly, hence we use the mechanism of separate enumeration (as previously described) to specify maximum parallelism.

As indicated earlier, in the Sisal program, the execution order of invocations of function sn is sequentialized by data dependencies. For example, a3 is available after a2 is available, therefore sn(a3) is executed after sn(a2). On the other hand, in the extended Sisal program the invocations of function gn can be executed concurrently; as explained in Section 4.2, each of the update operations old x + v can be viewed as a separate thread, and executed with a parallel coordination primitive. Therefore, the execution time of the Sisal program is more than that of the extended Sisal program.

Let us assume that one addition operation consumes one execution time unit. The execution time for the Sisal program is about 43 units, and the execution time for the extended Sisal program is about only 15 units. Since the function f can be executed in an arbitrary serial ordering, the execution results of main can be one of the following 10! cases: *values[101, 102, 103, 104, 105, 106, 107, 108, 109, 110]*, *values[102, 101, 103, 104, 105, 106, 107, 108, 109, 110]*,..., and *values[101, 102, 103, 104, 105, 106, 107, 108, 110, 109]*. The performance improvement, the ratio of execution time of the Sisal program over the extended Sisal program is about $\xi_{nn} = 43/15 = 2.8$. In addition, if the names are used by some other functions then the ratio would be much larger. Note that the order of the names is unimportant, and that extended Sisal allows programmers to make use of this property to exploit more parallelism.

6.2. Parallel Branch and Bound

Parallel branch and bound techniques can be applied to many applications. One example using this technique is known as the travelling salesman problem. The problem is to determine the length of the shortest path from source to destination.

Nodes are connected with edges that can be assigned costs: distance between two nodes connected by the edge (Fig. 7). We assume that all the costs are positive values. The length of a path is defined as the sum of the costs of the edges on that path. The starting node of the path is referred to as the *source* (node N_1), and the last node the *goal* (node N_4). We aim at the shortest path between the source and the goal. In the example, the shortest path is N_1-N_2-N_4, and the minimum value 15.

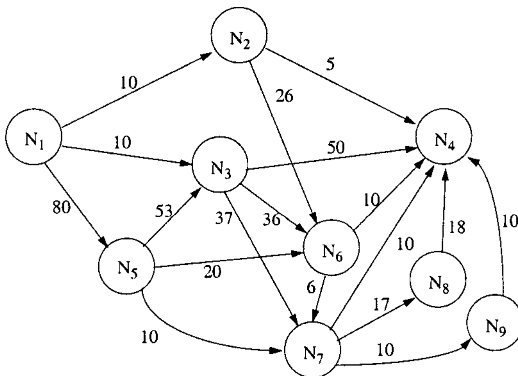

Fig. 7. Example of a graph for shortest path.

```
constant NoPath = 10000;
type Info = array of array of integer;
function main(returns integer)
        function f(Data: Info, L1, source, goal: integer
                       returns integer)
         if source = goal
           then L1
           else
            let A1:=
              for N in 2,9
                do L2:= L1+Data[source,N]
                returns
                  array of
                    if L2 >= NoPath
                    then NoPath
                    else f(Data,L2,N,goal)
                    end if
              end for
            in minval(A1)
            end let
         end if
        end function
 let
   COST : Info := array integer [1..9,1..9:
          [1,2] 10; [1,3] 10; [1,5] 80; [2,4] 5; [2,6] 26;
          [3,4] 50; [3,6] 36; [3,7] 37; [5, 3] 53; [5,6] 20;
          [5,7] 10; [6,4] 10; [6,7] 6; [7,4] 10; [7,8] 17;
          [7,9] 10; [8,4] 18; [9,4] 10;  [otherwise] NoPath];
 in  f(COST,0,1,4)
 end let
end function
```

Fig. 8. Shortest path in Sisal.

In pure Sisal, the application cannot be programmed by a shared variable but has to be purely functional (see Fig. 8).

A recursive call is performed for all valid paths:

```
if L2 >= NoPath
then NoPath
else f(Data,L2,N,goal)
```

A minimum is taken (in minval(A1)), from the local paths emanating from a given node, after the recursive calls complete.

Each process must collect all the returned values for each iteration of the loop, and decide the minimum value for these returned values. The

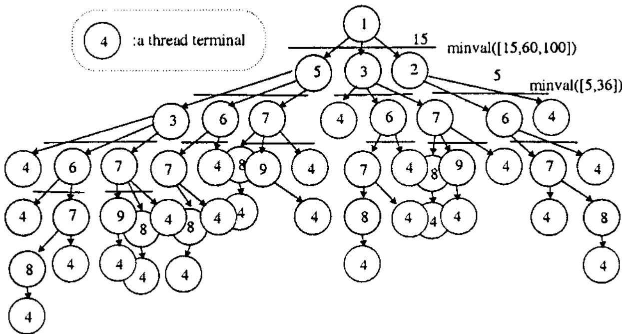

Fig. 9. Execution tree for shortest path in sisal.

```
constant NoPath = 10000;
type Info = array of array of integer;
function main(returns integer)
        function f(Data: Info, L1, source, goal: integer
                       returns integer)
           state Gmin : integer
              initial
                 Gmin := NoPath
           end state;

           if source = goal
           then let  Gmin := if L1 < old Gmin
                             then L1 else old Gmin end if
                in    L1
           else
            let A1:=
              for N in 2,9
                do L2:= L1+Data[source,N]
                returns
                  array of
                    if L2 >= NoPath or L2 >= Gmin
                    then NoPath
                    else f(Data,L2,N,goal)
                    end if
              end for
            in minval(A1)
            end let
           end if
        end function
 let
   COST : Info := array integer [1..9,1..9:
          [1,2] 10; [1,3] 10; [1,5] 80; [2,4] 5; [2,6] 26;
          [3,4] 50; [3,6] 36; [3,7] 37; [5, 3] 53; [5,6] 20;
          [5,7] 10; [6,4] 10; [6,7] 6; [7,4] 10; [7,8] 17;
          [7,9] 10; [8,4] 18; [9,4] 10; [otherwise] NoPath];
 in  f(COST,0,1,4)
 end let
end function
```

Fig. 10. Shortest path in extended Sisal.

entire graph must be traversed. Finding a minimum can be done in an order of time $O(N)$, where N is the number of elements in the COST array.

On the contrary, in extended Sisal, it can be implemented by a shared variable M as shown by the program in Figs. 9 and 10.

This variable is updated as soon as a new path to the goal is found, if it is better than the best previously found:

```
if source = goal
then let Gmin := if L1 < old Gmin
                 then L1 else old Gmin end if
   ...
```

Then, at each node of the graph during the search, a recursive call is done only if the current value is not greater than the current shortest path:

```
if L2 >= NoPath or L2 >= Gmin
then NoPath
else f(Data,L2,N,goal)
end if
```

A process does not need to wait until the entire iteration is finished before it updates M in the extended Sisal program. Collecting data into an array

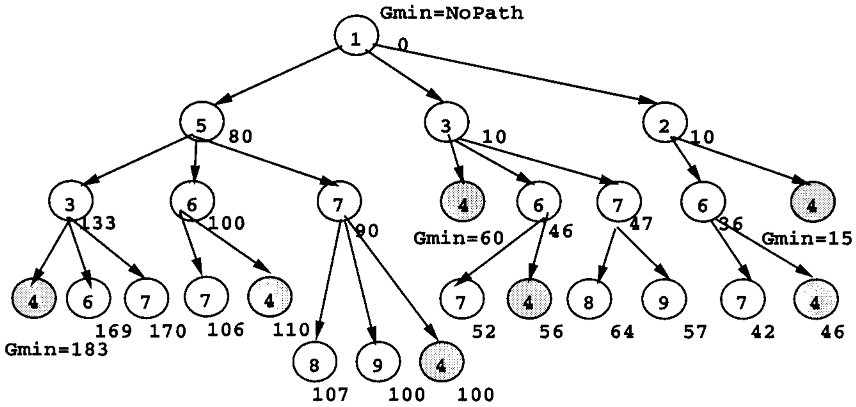

Fig. 11. One possible execution tree for shortest path in extended Sisal.

is not needed since the path value of each iteration is stored in M, if it is the shortest found so far. The decision of minimum value for the intermediate results of M has been scattered into all the iterations. Whenever there is a current path (it does not need to be complete), it is compared with the current M.

This eliminates most of the searching time and improves the performance, since the comparison is amortized and hidden in the concurrently executing processes.

An execution tree for the Sisal program is shown in Fig. 9, and an execution tree for the extended Sisal program is shown in Fig. 11.

We make the following hypotheses in order to reduce nondeterminism and restrict attention to one possible execution:

- no limits on the computational resources (we are actually using 14 threads or processes at most);
- processes are activated in parallel from left-to-right in the figure (node 5, then 3, then 2). For a given node, its sons finish after its right siblings. The labels Gmin = x denote the updates of the state variable Gmin (in order 183, 60, 15).

The degree of parallelism for Fig. 9 is greater than that of Fig. 11. However, the execution time is the real concern for programmers. The execution tree in Fig. 11 contains almost only useful operations, while the tree in Fig. 9 includes many nonessential operations, because in the Sisal program, function f sees information about only that sub-tree for which it is responsible; by using Extended Sisal, we can readily make the global minimum visible, and curtail further searching if the local minimum is

already more expensive than the minimum value computed for another sub-tree (and recorded in the global minimum). Again, this is an example of a situation where the order of updates to the shared value is not important, and we can profitably and safely use the techniques described earlier.

In addition, its standard execution also occupies more resources (storage locations, processors) than the extended Sisal program since there are more active processes executed.

6.3. Histogramming

This section presents an histogramming application: it counts the number of elements with a certain value in an array. Again we compare the standard Sisal program with a stateful version.

First of all, let us note that Sisal has two forms of loops. The "product form" is a parallel loop where there are no data dependencies between iterations; all of them can potentially be executed in parallel. The "non-product form" defines a sequential loop, in which certain loop variables are carried between iterations, and updated from one to the next; the simple loop of Fig. 12 is such a sequential loop. The two can be distinguished syntactically by the presence of an update to a loop-carried variable, such as n in Fig. 12.

Figures 12 and 13 respectively present the standard and stateful Sisal versions of the histogram program. In both cases the general algorithm is the same: go through the entire original array (data), and for each value encountered increment a counter at that index in a result array (result).

The most important difference between the two versions of the function histogram is that, in Sisal, accumulation into histogram elements must be expressed as a sequential loop (for ... while i <= InfoSize do ...),

```
type Info = array of integer;
constant InfoSize = 1000;   % size of data array
constant InfoValues = 10;   % data values are in 1 .. InfoValues

function histogram (data : Info returns Info);
  for
    histo := array integer [1..InfoValues : [1..] 0);
    i := 1;
  while i <= InfoSize do
    j     :=   data[i];
    i     :=   old i + 1;
    histo :=   old histo [j : old histo[j] + 1];
  returns histo;
  end for;
end function;

function main returns Info;
let
  data: Info := array integer [1..InfoSize: 1, 2, 6, 1, 7, ...];
  result : Info  := histogram (data);
in
  result
end let;
end function;
```

Fig. 12. Histogramming in Sisal.

```
type Info = array of integer;
constant InfoSize = 1000;   % size of data array
constant InfoValues = 10;   % data values are in 1 .. InfoValues

function histogram (data : Info returns Info);
  function histo_state (j : integer returns Info );
    state    histo : Info
      initial  histo := array integer [1..InfoValues : [1..] 0);
    end state;
    let   histo := old histo [j : old histo[j] + 1];
    in    histo
    end let;
  end function;
  for e in data at i do
     hs := histo_state (e)          % e = data[i]
  returns value of hs;
end function;

function main returns Info;
let
  data: Info := array integer [1..InfoSize: 1, 2, 6, 1, 7, ...];
  result : Info  := histogram (data);
in
  result
end let;
end function;
```

Fig. 13. Histogramming in extended Sisal.

whereas in Extended Sisal we can use a parallel loop, fully parallelized over the elements of the data array (for e in data at i do ...). Indeed, it would be very hard, if at all possible for the general case, to express a parallel version of the histogram algorithm in Sisal. The underlying reason being, in that category of problem, the accumulator array produces a global data dependency that leads to a synchronization on the entire array.

On the contrary, for the stateful Sisal version, the loop is parallel because the accumulation occurs in a stateful function that was added for that purpose (histo_state). Thus, the iterations become independent. Each iteration of the loop indirectly defines an update in the histogram; the iterations are sequentialized dynamically only within the constraint of mutual exclusion on elements of the histogram array itself. We went from a data dependency on the entire accumulator array precluding all parallelism, to data dependencies at the level of each cell of the accumulator array. Clearly, this provides much greater potential parallelism.

7. CONCLUSIONS

Finding parallelism in imperative programs is difficult because the class of programs that are amenable to data dependence analysis is limited. Functional languages remove some of the burden of managing parallelism from the programmer, but some types of computations are difficult to express without state. Moreover, we have shown that the functional expression sometimes actually limits parallelism. By extending Sisal with the concept of state, we have developed a language whose semantics fits usefully between a single-assignment functional language and an imperative language. The extension enhances programmability in many cases where it

is difficult to express maximal parallelism in a conventional Sisal encoding of an algorithm; in these cases, programmability is improved in that state functions allow greater parallelism to be expressed for minimal coding effort, certainly less than would be required to express the same parallelism in conventional Sisal.

We have presented in this paper a specification of an extension to the Sisal functional language. This specification uses the Centaur system.[23] The presence of side-effects is usually regarded as harmful, but some actions leading to side-effects can sometimes be considered useful for several parallel applications. We have thus extended the syntactic and semantic specifications of Sisal[8, 9] to formally define our stateful extension. The METAL and TYPOL rules used to specify the syntax and semantics of extended Sisal have been described. An interactive programming environment for extended Sisal has been generated from its formal specifications and several extended Sisal programs have been developed to demonstrate the usefulness of this language.[6, 7]

We provide programs in both extended Sisal and in Sisal to demonstrate that with the state variable scheme it is easier and more efficient to program parallel computing for appropriate applications. The main contributions of our work are as follows: programmers can express stateful computations in a language supporting functional parallel multiprocess execution. The stateful features in programs can be easily recognized and parallelism can be increased: stateful aspects are syntactically distinguished so that non-functional parts of a program are clearly delineated. Users can remain in a purely functional language style when its simplicity and implicit parallelism are desired. In comparison with purely functional languages, the extended Sisal has greater expressive power, especially in terms of parallel processing. We believe that these advantages can be found useful in many applications in addition to the examples in this paper.

ACKNOWLEDGMENTS

This research was supported in part by the NSF/INRIA/CNRS grant #CSA-0073527 and INT-9501081.

REFERENCES

1. J.-L. Gaudiot and C. Kim, Data-Driven and Multi-Threaded Architectures for High-Performance Computing. In T. Casavant and P. Tvrdik, (eds.), *Parallel Computers: Theory and Practice*, Chap. 4, IEEE Computer Society Press, Washington, D.C. (1993).
2. J. L. Gaudiot and Y. Wei, Token Relabeling in a Tagged-Token Data-Flow Architecture, *IEEE Trans. Computers*, **38**(9) (September 1989).

3. P. Hudak. In B. Szymanski, (ed.), *Para-Functional Programming in Haskell*, ACM Press, New York (1991).
4. P. Hudak, Functional Programming Languages, *ACM Computing Surveys*, **21**(3):359–411 (September 1989).
5. A. P. W. Bohm, R. R. Oldehoeft, D. C. Cann, and J. T. Feo, Sisal Reference Manual Language Version 2.0, Computer Science Department, Colorado State University, and Computing Research Group, Lawrence Livermore National Laboratory (1990).
6. Y. S. Chen and J. L. Gaudiot, Semantics Specifications of Extended Sisal 2.0 in the Centaur System, *Proc. Eighth IASTED Int'l. Conf. Parallel and Distributed Computing Syst.*, pp. 410–414 (October 1996).
7. Y. S. Chen and J. L. Gaudiot, An Application of Extended Sisal, *Proc. Int'l. Conf. Parallel and Distributed Processing Techniques and Applications*, pp. 245–251 (August 1996).
8. I. Attali, D. Caromel, and A. Wendelborn, A Formal Semantics and an Interactive Environment for Sisal. In A. Zaky (ed.), *Tools and Environments for Parallel and Distributed Systems*, Kluwer Academic Publishers, pp. 231–258 (February 1996).
9. I. Attali, D. Caromel, Y. S. Chen, J. L. Gaudiot, and A. Wendelborn, A Formal Semantics for Sisal Arrays, *Proc. Joint Conf. Infor. Sci.* (September 1995).
10. S. Peyton Jones and P. Wadler, Imperative Functional Programming, *ACM Principles of Progr. Lang.* (1993).
11. J. Launchbury, Lazy Imperative Programming, Technical Report, Department of Computer Science, University of Glasgow (December 1993).
12. P. Wadler, The Essence of Functional Programming, *ACM Principles of Progr. Lang.* (1992).
13. H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, G. L. Steele Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand, Revised^5 Report on the Algorithmic Language Scheme, *J. Higher Order and Symbolic Computation*, **11**(1):7–105 (1998).
14. R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*, MIT Press (1997).
15. R. S. Nikhil and Arvind, Id: A Language with Implicit Parallelism. In J. Feo, (ed.), *Comparative Study of Parallel Programming Languages: The Salishan Problems*, Elsevier Science Publishers (1990).
16. Arvind and R. Nikhil, *I*-structures: Data Structures for Parallel Computing, *ACM Trans. Progr. Lang. Syst.*, **11**(4):598–632 (October 1989).
17. K. Pingali and K. Ekanadham, Accumulators: New Logic Variable Abstractions for Functional Languages, *Theoret. Computer Sci.*, **81**(2):201–221 (April 1991).
18. P. S. Barth, R. S. Nikhil, and Arvind, *M*-Structures: Extending a Parallel, Nonstrict, Functional Language with State, *Proc. Fifth Conf. Functional Progr. Lang. Computer Architecture*, Cambridge, Massachusetts, LNCS 523, Springer-Verlag (August 1991).
19. S. Peyton Jones and L. Duponcheel, Composing Monads, Technical Report Research Report YALEU/DCS/RR-1004, Yale University (December 1993).
20. P. Wadler, Comprehending Monads, *Proc. ACM Conf. Lisp and Functional Progr.*, Nice (1990).
21. G. Almasi and A. Gottlieb, *Highly Parallel Computing*, Second Edition, Benjamin/ Cummings (1994).
22. Burton Smith, The Tera Computer System, *Proc. ACM Int'l. Conf. Supercomputing*, pp. 1–7 (1990).
23. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, Centaur: The System, *Proc. SIGSOFT'88, Third Ann. Symp. Software Dev. Environments*, Boston (1988).

24. S. Skedzielewski and J. Glauert, IF1—An Intermediate Form for Applicative Languages, Manual M-170, Lawrence Livermore National Laboratory, Livermore (1985).
25. M. L. Welcome, B. K. Szymanski, R. K. Yates, J. E. Ranelletti, An Applicative Language Intermediate Form Explicit Memory Management, Manual M-195, Lawrence Livermore National Laboratory, Livermore (1986).
26. G. Kahn, Natural Semantics, *Proc. Symp. Theoretical Aspects of Computer Science*, Passau, Germany, LNCS 247 (February 1987).
27. I. Attali, D. Caromel, M. Oudshoorn, A Formal Definition of the Dynamic Semantics of the Eiffel Language, *16th Australian Computer Sci. Conf. (ACSC-16)*, Brisbane, Australia, 1993, also Research Report I3S 92.52.
28. A. Berstein, Analysis of Programs for Parallel Processing, *IEEE Trans. Computers*, pp. 746–757 (1966).
29. Y. S. Chen and J. L. Gaudiot, Parallelism Detection Algorithm for Extended Sisal Programs in Centaur, *Proc. Eight ISCA Int'l. Conf. Parallel and Distributed Computing Systems*, pp. 628–633 (September 1995).
30. Y. S. Chen and J. L. Gaudiot, Extending Functional Languages with Stateful Computations, *Proc. Eight IEEE Symp. Parallel and Distributed Processing*, pp. 542–549 (October 1996).
31. T. DeBoni, J. Feo, and D. Peters, Integrating Imperative and Functional Programming in Real World Applications, *Proc. High Performance Functional Computing* (April 1995).
32. D. Engelhardt and A. Wendelborn, Investigating the Memory Performance of the Optimizing Sisal Compiler. In J. Feo, C. Frerking, and P. Miller, (eds.), *Proc. Second Sisal Users' Conf.*, Lawrence Livermore National Laboratory, pp. 257–270 (December 1992).
33. S. M. Fitzgerald, Copy Elimination for True Multidimensional Arrays in Sisal 2.0, *Proc. Third Sisal Users and Dev. Conf.*, San Diego, California (October 1993).
34. J.-L. Gaudiot, Structure Handling in Data-Flow Systems, *IEEE Trans. Computers*, **35**(6):489–502 (June 1986).
35. E. Horowitz and S. Sahni, Fundamentals of Data Structures in Pascal, Computer Science Press, New York (1989).
36. K. Hwang, *Advanced Computer Architecture with Parallel Programming*, McGraw–Hill Publishing Company, Inc., New York (1993).
37. C. Kim, J.-L. Gaudiot, and W. Proskurowski, Programmability and Performance Issues: Case of an Iterative Partial Differential Equation Solver, *Proc. Sisal '93*, San Diego, California (October 1993).
38. B. Meyer, *Introduction to the Theory of Programming Languages*, Prentice–Hall, Inc., Englewood Cliffs, New Jersey (1991).
39. P. Wadler, Monads for Functional Programming, *Lecture Notes for Marktoberdorf Summer School on Program Design Calculi*, Springer-Verlag (August 1992).
40. A. Wendelborn and H. Garsden, Exploring the Stream Data Type in Sisal and Other Languages. In M. Cosnard, K. Ebcioğlu, and J.-L. Gaudiot, (eds.), *IFIP Transactions: Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, IFIP, North-Holland, pp. 283–294 (January 1993).

Printed in Belgium