Stateful Computations in Functional Languages

Yung-Syau Chen

CENG-96-10

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4484

May 1996

STATEFUL COMPUTATIONS IN FUNCTIONAL LANGUAGES

by

Yung-Syau Chen

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

May 1996

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90007

This dissertation, written by

YUNG-SYAU CHEN

under the direction of h...is... Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of

DOCTOR OF PHILOSOPHY

Dean of Graduate Studies

Date April 18, 1996

DISSERTATION COMMITTEE

Chairperson

Dedication

To my wife Josephine. With your love and encouragement, I am able to finish the program.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Jean-Luc Gaudiot, for his generous guidance, encouragement, and support throughout these years of my graduate study. It is a great pleasure working with him. I am extremely lucky to have him as my advisor.

I am very grateful to Professor Sandeep Gupta and Ellis Horowitz for serving on my dissertation committee. Their inquisitive questions have greatly stimulated my research. I would also like to thank Professors Kai Hwang and Victor Prasanna for being on my Ph.D guidance committee.

I am very grateful to Dr. Isabelle Attali, Dr. Denis Caromel, and Dr. Andrew L. Wendelborn who have shown great interest and have given us great help in our work. I also highly appreciate Wes Hansford, Shih-Lien Lu, Jen-I Pi, Jeff Sondeen, Vance Tyree, and Tzyh-Yung Wuu for their help during the time I worked in MOSIS/ISI.

Valuable discussions with graduated doctoral colleagues from PDPC (Parallel and Distributed Processing Center) were truly helpful, including Dr. Chinhyun Kim, Dr. Chin-Ming Lin, and Dr. Andrew Sohn. I would like to thank my group members Moez Ayed, Hiecheol Kim, Daekyun Yoon, and Namhoon Yoo for friendly interaction. Numerous discussions I had with them have stimulated my research greatly. Many thanks to group members Chung-Ta Cheng, Jim Burns, Halima Elnaga. Steve Jenks, Wen-Yen Lin, Chulho Shin, and Hung-Yu Tseng for their continuous assistance and warm friendship. I am so lucky to have them as my group members. Special thanks goes to Mary Zittercob, Rohini Montenegro, Joanna Wingert and Kiet Phu for their nice assistance.

I thank my parents for raising and educating me. The encouragement from them and my uncles, aunts, brothers, and sisters motivated me very much during the course of my Ph.D. study.

Contents

List Of Figures

Abstract

We present a new approach in which stateful computations can be performed within the framework of a functional programming language supporting parallel multi-threaded execution. Efficient use of parallel machine is only feasible when applications can be expressed without loss of their inherent parallelism.

To extract maximum parallelism from many applications, programmers need stateful computations. A great deal of work has been carried out on functional languages that guarantee the output of a program to be a function of its input. However, in most functional programming languages, programmers are unable to easily manipulate state-based computations since they are not supported by functional languages. To solve this problem, we propose to extend the Sisal language with special user-declared variables. Sisal is a functional language which has been designed by a consortium of industrial and research organizations for the specification and execution of parallel programs.

Our approach can greatly help users in writing programs, simplifying parallel compilation, and improving performance. Under this scheme, (1) programmers will be able to manipulate stateful computations, (2) the expressions which can be evaluated in parallel will be easily identified, and (3) higher degree of parallelism can be delivered.

In our methodology, programmers are allowed to declare special variables, and the expressions which must be executed in sequence can be identified according to the usage of special variables. In comparison to purely functional languages, extended-Sisal has more expressive power due to the availability of stateful computations. We

demonstrate that this approach can greatly improve the performance of a parallel processing system.

Chapter 1

Introduction

The problem of programming parallel computers is still without a definite solution [1]. On the low-level side, driven by the high cost of complex data-flow architectures, the execution model for exploiting fine- and medium-grain parallelism has evolved gradually from the data-driven to the multi-threaded execution model [2] [3]. Functional programming languages have been used in writing programs for parallel multi-threaded systems because they can extract the maximum potential parallelism from applications. On the high level language front, the explicit advantages of functional languages have always been "programmability" and the easy extraction of parallelism [4] [5]. Indeed, the processing power of thread-based multiprocessors is useful only when the application, the programming language, and the compiler can effectively take advantage of it. Provided with higher-level abstractions and the implicit parallelism of functional languages, programmers using functional languages can concentrate on the implementation of the algorithms without being concerned with low level execution details such as synchronization and communication [6]. However, for algorithms with inherent stateful computations, the expressive power of functional languages is insufficient. We intend to provide the means to overcome the limitations imposed by current functional programming languages.

The goal of our work is to demonstrate how Sisal can be extended with stateful computations to improve the programmability and performance for real world applications while keeping the ease of parallel execution which comes along with the nature of functional languages.

The rest of the dissertation is organized as follows. Chapter 2 compares the imperative and functional programming languages in terms of delivery of parallelism. Chapter 3 reviews previous work. Chapter 4 gives the motivation for our work. Our methodology is presented and discussed in Chapter 5. The verification work which uses the Centaur system is described in Chapter 6. Programs for some parallel applications and their performance evaluation in both regular and extended-Sisal are presented in Chapter 7. Finally, conclusions are given in Chapter 8.

Chapter 2

Comparisons of Programming Languages

It is generally impossible to illustrate the absolute superiority of one language over another in all conceivable cases. For instance, high level programming languages seem better than assembly languages in most occasions, but there are cases in which low level assembly languages are still the preferred approaches to programming. What can be argued is that some languages are more useful for a large collection of practical applications. Programming languages are widely divided into "imperative" and "functional" types. In an imperative language program, a variable is a quantity whose value can be obtained from a particular storage location. The value in this storage location can be referenced and updated by the program. This mutable shared variable notion corresponds to the globally accessible location and provides a useful mechanism for sharing data structures. This also provides a convenient way for communication between various parts of the program. However, this feature makes parallelism extraction very difficult. On the other hand, in a functional program, the concept of variable is similar to its mathematical sense. Typically, a function takes some variables and produces new variables while the value of any individual variable is never changed.

2.1 Referential Transparency

Referential transparency is the basic difference between functional and imperative languages. It is a fundamental property of mathematical notations, and has first been described for propositions by Whitehead and Russell [7]. Referential transparency is simply defined as: two expressions are equal if they indicate the same value. An immediate consequence of this definition is that the value of a composite expression depends only on the values of its constituent expressions [7]. Because of this property, it is easier for an execution system to perform parallel evaluation of functional programs.

2.2 States and Assignments

A practical model employed by scientists, communication engineers, and circuit designers is the state-transition system [8]. Computing scientists use state-transition models in studying formal languages. Communication engineers represent communication protocols as state-transition systems. One way to manipulate the state-transition operations while enjoying the representational advantages of programming languages is to use assignments [9]. In a pure functional language, the locality of effect must be enforced by making assignments only to local variables. This means that in these languages, global assignments and common variables cannot be used [10]. The easiest way to avoid the accessing a wrong version of data is to prevent performing destructive updates on variables since only destructive updates cause multiple values of variables. This approach is adopted by single assignment functional languages such as Sisal, Miranda, Haskell, etc. A single assignment functional language prohibits destructive updates. The single assignment rule means that a data structure must be copied every time an assignment is performed [11]. In a program, an identifier can serve as explicit store for a value and can only be assigned once. The non-destructive update requires more storage locations and execution

time since the unchanged information from the original data structure has to be copied into a new one.

2.3 Imperative Programming Languages

Imperative programs are sequences of *commands*. The term imperative is used to describe a program or programming language which computes by effect rather than value only [12]. An imperative language program proceeds by repeatedly computing a value and assigning it to a location in memory. Because this behavior closely matches to the von Neumann architecture, such programs are extremely efficient on conventional machines. The principal features of writing an imperative language are: (1) variables are considered to possess values which can be changed by assignments, (2) the way a program is computed is by making a sequence of changes to the values of variables, and (3) the new value for each variable is determined by the current values of some variables in the program.

2.3.1 Storage Oriented

Imperative programming languages are storage oriented. The center of these languages is the storage location that stores everything needed for computations at a globally accessible address. The commands update variables held in memory locations, and therefore an imperative program is said to have a state that is modified when the program progresses. One of the difficult things about imperative programming languages is that they are not well suited to parallelism extraction. Since imperative languages are oriented towards the sequential machine model, they are not easily compiled for execution on parallel machines [13]. Due to their maturity, imperative languages such as FORTRAN, C and Pascal is normally the languages of choice for computers today. While these languages have gained wide acceptance in commercial and scientific contexts [14], none were ever intended to operate on

parallel execution environments. Therefore, it is not a simple task for programs written in imperative languages to exploit much potential parallelism from practical applications.

2.3.2 Limitations

Due to the nature of imperative languages, compilers for imperative languages may miss a lot of potential parallelism. Usually more information must be made available to the parallelizing compiler by programmers so that further optimization could be performed to identify more potential parallelism [14]. To run a program in parallel, the compiler for imperative languages have to distinguish real data dependencies from the artificially imposed dependencies caused by the nature of the languages. The opportunities for parallelism are often not exploited because the compiler is forced to generate sequential code when there is no information about whether specific code fragments should be running in parallel or not. Designing imperative programs for multi-threading is difficult because many synchronization and communication details have to be taken care of besides programming the algorithm. This can greatly reduce the programmability and affect the programmer's ability to write correct and efficient programs, and it is difficult to extract the parallelism without imposing complicated synchronization mechanisms.

Suppose that programmers want to get a FORTRAN program to run on a parallel computer, they must themselves implement a small set of start/stop primitives and synchronization primitives. As a result, the parallelism is created manually by programmers. One example is the set of primitives of the NYU Ultra-computer [15]. Primitives like *do-all* and *par-begin* are implemented as actual language extensions recognized by a preprocessor to the compiler. Some primitives are library routines that are called by the program.

Some earlier work has been performed on parallelism extraction from sequential imperative programs. Many attempts have been made to extend existing imperative

languages with constructs for explicitly specifying parallelism [1]. With a parallelizing compiler, programmers do not have to modify the program themselves. However, with this approach the performance improvement is frequently small, and designing such compilers is very complicated. Consequently, explicitly parallel variants for imperative languages have been designed. Such explicit parallelism constructs are more broadly available than automatic parallelizing compilers. The disadvantage is that programmers need to take care of the detail of the parallelizing.

2.4 Functional Programming Languages

Functional programming languages attempt to extract the maximum possible parallelism from an application and make the synchronization easy to implement [16] [5]. These languages grew out of the belief that the basic features of the imperative programming languages reduced programmer productivity and parallelism exploitation. One of the best features of functional programming languages is that they are well suited to parallelism extractions. Functional languages share many features with logic languages such as Prolog, including their declarative nature and potential for parallelism. Further, in a functional program, the modes of input and output arguments are fixed, while in a logic program, they are query-dependent [17]. There are several machines, and more under development, which are designed to evaluate functional languages in parallel [7]. Generally, a functional programming language is different from an imperative language because it is concerned with expressing what a program does rather than how it does it. In other words, it is more *user-oriented* than *system-oriented.* In contrast to an imperative language, a functional language contains no side-effect or implicit state that can be modified by assignments. The first functional programming language was invented by A. Church during the 1930 through lambda calculus [18]. At that time, there were no computers to run the programs. The development of computer set off the development of a series of imperative

programming languages suited for particular computers: FORTRAN, COBOL, Algol, Pascal, and so on. An increasing discomfort with these imperative languages resulted in the development of a series of functional programming languages: FP, pure-Lisp, Scheme, Hope, Sisal, Miranda, and so on. One of the reasons for the development of these functional programming languages has been conceptual simplicity.

Programs written in a functional language can be easily translated into data flow graphs to identify the maximum parallelism in the application. In a data flow graph, nodes represent functions and arcs represent data dependencies. Driven in part by the high cost of complex data-flow architectures, the execution model for exploiting fine- and medium-grain parallelism has evolved gradually from the data-driven to the multi-threaded execution model [2] [3]. Functional programs do not carry the concept of implicit state or memory locations; an ordinary variable in a functional language program does not correspond to one particular memory location. In FP proposed by Backus [19], computations are carried out entirely through the evaluation of expressions. Purely functional languages such as FP have proved to be neither suitable to efficient execution nor sufficiently rich in language features. Removal of side-effects results in a language that needs more storage capacity and bandwidth. The extra storage is needed for all the elements that are copied rather than rewritten. If the compiler cannot delete superfluous copying, that would result in low performance.

Sisal is a functional language which has been demonstrated to have a performance comparable with FORTRAN on a number of computing platforms [20] [21]. It allows programs to be written with little concern for the structure of the underlying machine, thus the programmer is free to explore different ways of expressing the parallelism [11]. Sisal can be compiled for a number of different machines and architectures. It prohibits the ability to express constructions leading to the side-effects which would make compilation for parallel systems difficult. For example, such statements as COMMON and EQUIVALENCE statements do not exist in Sisal.

Haskell is also a general purpose single assignment functional language. It contains many characteristics of other languages and is intended to provide a base for communicating ideas, developing practical applications, etc. [22]. However, Haskell has no constructs inducing side effects to an implicit store. Since single assignment functional languages lack a global state, writing programs for applications which involve stateful computations is usually very complicated. We will show that this problem can be solved by the extension of the single assignment functional language with semantics of stateful computations.

Chapter 3

Previous Work

There have been several attempts to combine the best features of functional programming languages and those of imperative languages [23]. Many researches have attempted to include stateful computations in a functional language and still maintain the ease of parallelism and clean properties of functional languages [24] [25] [26]. In this chapter, some of the most recent approaches to stateful computations in functional programming languages will be discussed.

3.1 Sisal

Sisal is a descendent of VAL and is the first functional programming language to perform as well as FORTRAN on scientific codes running on an 8-processor Cray Y-MP. Sisal is a functional language with no explicit parallel control constructs. The data types and its constructs to express operations are specifically selected to generate efficient programs for large-scale scientific computations [27]. In a Sisal program, no global variables are allowed, and variables cannot be multiply assigned. [28] proposes to allow a Sisal program to use codes written in a foreign language. This approach is promising and deserve an investigation. However, The approach requires a Sisal interface to foreign codes for nonexistent associated modules [29]. The idea behind this approach is that a functional program cannot perform stateful

computations, and that outside imperative programs should instead do it. This approach requires the outside compiler to be modified.

3.2 Id

Id has been extended with I-structures, which made the language nonfunctional without losing determinacy [30]. While I-structures solve some of the problems which arise from functional data structures, programmers have frequently encountered another class of problems for which there are no efficient solutions [30]. One example of a problem which I-structures cannot solve is *histogramming*. Therefore, Id has been extended to include the *accumulator operator*. More recently, Id has been further augmented with M-structures, which allow multiple assignments. The reason for including M-structures is to solve the application containing the structure like a descriptor working queue. M-structures are indeed multiply re-assignable structures. Figure 3.1 illustrates these four layers of constructs in Id. In the pure functional core, the results of functions can usually be returned as physical parameters to another function. One example of the applicative constructs in the functional core layer is shown as follows:

```
   plus (2*3) (plus 2 3)
=> plus 6 (2+3)
=> plus 6 5
```

There are no variable assignments in this example, since the results of functions can usually be directly returned as physical parameters to another function. In the following example with I-structures, a variable B is assigned to an I_array. Each element in the array can only be assigned once. An array element can be read whenever it is available, even though the remaining elements in the array are still not available:

```
B= I_array(1,N);
if (A[j]>=0) then B[j]=A[j]
```

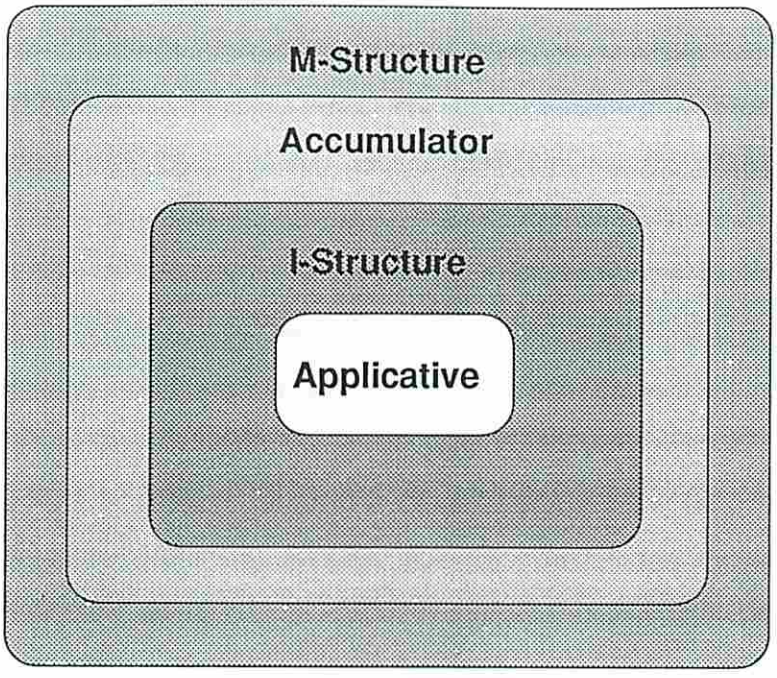

Figure 3.1: Program construct layers of Id

```
else B[j]=0
```

I-structure cannot represent applications like histogramming, for which Id uses *accumulators*. The operator in an accumulator, such as "+" in the following example, must be associative and commutative:

```
{array (1,5) of
 | [j]= 0 || j <- 1 to 5
 accumulate (+)
 | [g]= 1 || i <- 1 to 200; g= grades[i]}
```

In this program, [j]= 0 initializes the array element indexed j, and [g]= 1 increases the value of the element indexed g. The results of the array can be read only after all elements are available. Within an accumulator the order of updating the element is unimportant. The results of the whole array can be read only after all inputs to the accumulator have been finished and all array elements are available. This is called a semi-strict construct. For representing applications like a descriptor

working queue, where one thread gets a value from a queue, works on it, and then returns a result to the queue, Id allows programmers to use M-structures which can be multiply assigned. In the following M-structure example, initially the value of A![j] contains a value, after the value is used, A![j] is assigned 3.

```
A= { M_array(1,5) of
  [j]= 0 || j <- 1 to 5}
  oldv= A![j];
  A![j]= 3;
```

Reads and writes to A![j] must be balanced, i.e., one read must be matched by exactly one write. The symbol "!" in A![j] denotes an M-structure data object.

3.3 Monads of State Transformer in Haskell

The concept of monads comes from the category theory. In recent years, it has become an important tool for functional programmers because monads provide a uniform framework for describing a wide range of programming features including I/O and states [31]. Much of the recent interest in monads in functional programming is the result of recent papers that have shown how monads can be used to describe the manipulation of state in functional languages like Haskell [24] [32] [26]. These papers explore the uses of monads to organize functional programs. Monads increase the ease with which programs can be modified. They can mimic the effect of impure features such as state [26]. Figure 3.2 illustrates one example of the monad state in Haskell [32]. For example, if

```
(A) (Lam x (x+2 (Lam x (3*x 4))))
```

becomes

```
(B) (Lam x0 (x0+2 (Lam x1 (3*x1 4))))
```

then the expression would be easier to understand. Applying formula (A) to the program in Figure 3.2 returns formula (B).

(1) newname = [mkname n | n <- fetch, () <- assign(n+1)]ST

(2) renamer (Var x) = [Var x] ST

(3)renamer(Lam x t) = [Lam x1 (subst x1 x t1))| x1<- newname, t1 <-renamer t]ST

(4) renamer(App t u) = [App t1 u1| t1 <-renamer t, u1<-renamer u]ST

(5) main t = init 0 (renamer t)

Figure 3.2: Program construct of monad state in Haskell

```
(main (Lam x (x+2 (Lam x (3*x 4)))))
=> (init 0 (rename (Lam x (x+2 (Lam x (3*x 4))))))
=> (Lam x0 (subst x0 x (rename (x+2 (Lam x (3*x 4)))))
=> (Lam x0 (subst x0 x (x+2 (Lam x1 (subst (x1 x (rename 3*x 4)))))))
=> (Lam x0 (subst x0 x (x+2 (Lam x1 (subst (x1 x (3*x 4)))))))
=> (Lam x0 (subst x0 x (x+2 (Lam x1 (3*x1 4)))))
=> (Lam x0 (x0 (x0+2 (Lam x1 (3*x1 4)))))
```

Clearly, in this application programmer need a global counter and that is implemented as an implicit state. However, it is difficult for the implicit state method to express more than one state (e.g., two counters.). In addition, the implicit state method is an abstract form of passing around parameters, and thus inevitably imposes an artificial serialization in the program even if the serialization is not inherent in the applications. Indeed, no order of new names is predetermined in the algorithm. For example, both of the following results are acceptable,

```
(1) (Lam x0 (x0+2 (Lam x1 (3*x1 4))))
(2) (Lam x1 (x1+2 (Lam x0 (3*x0 4))))
```

but the above program always returns (1).

Chapter 4

Motivations

There are certain components of parallel algorithms which cannot be easily expressed by pure functional languages because of their lack of stateful computations. Since the single assignment rule limits the choice of program structures, so far, Lisp and Prolog are the only examples of "less imperative" languages which have achieved wide use. However, Lisp or Prolog programs that actually perform something other than the toy examples seem to make use of a lot of imperative features. These imperative features include PROG, SETQ, RPLACA of Lisp and the cut mechanism in Prolog [33]. In Lisp, PROG is doing explicit sequencing, SETQ is doing assignments, and RPLACA is associated to side-effects on data structures. In Prolog, the cut mechanism corresponds to non-regular sequencing.

It is well believed that functional languages will become more widely used in the future, but many programs which really do something useful are those which contain some imperative natures.

So how to solve this dilemma, without giving up the functional programming paradigm which has many other advantages? We introduce an extension to the functional programming paradigm, which allows users to express stateful computations without losing most of the well-known benefits of functional languages. Many problems caused by the limitation of expressive power of functional languages can be solved by allowing the update of particular variables. Programmers should be able

to manipulate stateful computations, and the imperative features in an application can be easily tracked.

4.1 The Bernstein's Conditions

A set of conditions based on which two threads can execute in parallel have been demonstrated by Bernstein [34] [35]. The *input set* I_i is defined as the set of all input variables needed to execute the threads T_i, and the *output set* O_i includes all output variables generated after the execution of thread T_i. Consider two threads T_1 and T_2 with their input sets I_1 and I_2 and output sets O_1 and O_2, respectively. These two threads can execute in parallel if the following conditions are verified:

$$\boxed{\begin{array}{l} I_1 \bigcap O_2 = \Phi \text{ (flow-independent condition)} \\ I_2 \bigcap O_1 = \Phi \text{ (anti-independent condition)} \\ O_1 \bigcap O_2 = \Phi \text{ (output-independent condition)} \end{array}}$$

These three equations are known as the *Bernstein's conditions.* They are used to detect parallelism as follows: *A set of threads,* T_1, T_2, ..., T_k, *are parallel threads if and only if the Bernstein's conditions are satisfied on a pairwise basis:* $\beta(P_i, P_j)$= *True for all* $i \neq j$. In the above, $\beta(T_1, T_2)$ means the Bernstein's conditions are satisfied for T_1 and T_2. The execution of two parallel threads produces the same results regardless of whether they are executed sequentially, in any order, or in parallel.

It will be shown that the Bernstein's conditions can be relaxed to extract higher degrees of parallelism. Consider the simple case in which each threads is a single statement. users want to detect the parallelism embedded in the following five instructions labeled T_1, T_2, T_3, T_4, and T_5.

$$
\begin{aligned}
&T_1 : C = f(G) \rightarrow I_1 = \{G\}, O_1 = \{C\} \\
&T_2 : D = C + g(H) \rightarrow I_2 = \{C, H\}, O_2 = \{D\} \\
&T_3 : F = A + C \rightarrow I_3 = \{A, C\}, O_3 = \{F\} \\
&T_4 : A = B + M \rightarrow I_4 = \{B, M\}, O_4 = \{A\} \\
&T_5 : D = A \div E \rightarrow I_5 = \{A, E\}, O_5 = \{D\} \\
&T_6 : L = N \div E \rightarrow I_6 = \{N, E\}, O_6 = \{L\}
\end{aligned}
$$

15 statement pairs must be checked against Bernstein's conditions. Assume that $\sim \beta(X, Y)$ means the Bernstein's conditions for X and Y are not satisfied. In the following $T_1 \perp T_2$ indicates that T_1 and T_2 must execute sequentially in the program order.

$$
\begin{aligned}
&\sim \beta(T_1, T_2) \Rightarrow T_1 \perp T_2 \text{ (violates the anti-independent condition)} \\
&\sim \beta(T_1, T_3) \Rightarrow T_1 \perp T_3 \text{ (violates the anti-independent condition)} \\
&\sim \beta(T_2, T_3) \Rightarrow T_2 \perp T_3 \text{ (violates the output-independent condition)} \\
&\sim \beta(T_2, T_5) \Rightarrow T_2 \perp T_5 \text{ (violates the output-independent condition)} \\
&\sim \beta(T_3, T_4) \Rightarrow T_3 \perp T_4 \text{ (violates the flow-independent condition)} \\
&\sim \beta(T_3, T_5) \Rightarrow T_3 \perp T_5 \text{ (violates the output-independent condition)} \\
&\sim \beta(T_4, T_5) \Rightarrow T_4 \perp T_5 \text{ (violates the anti-independent condition)}
\end{aligned}
$$

Consequently, there are seven ordering restrictions. Suppose that D, F, and L are special variables, which are declared by the user, and A and C are ordinary variables, which can be assigned only once. For A and C, none of the flow-independent, anti-independent, and output-independent conditions need to be checked since data availability principle is sufficient. Threads T_1 and T_2 can execute out of order, sequentially, or in parallel. Therefore, functions f(G) and g(H) can execute concurrently. Special variables, however, need to satisfy these three conditions. In the following the *reduced input set* and *reduced output set* for a threads are denoted by I' and O', respectively. O' is the same as update_iset introduced in [36].

$$
\begin{array}{|l|}
\hline
T_1 : C = f(G) \rightarrow I'_1 = \{G\}, O'_1 = \{\} \\
T_2 : D = C + g(H) \rightarrow I'_2 = \{H\}, O'_2 = \{D\} \\
T_3 : F = A + C \rightarrow I'_3 = \{\}, O'_3 = \{F\} \\
T_4 : A = B + M \rightarrow I'_4 = \{B, M\}, O'_4 = \{\} \\
T_5 : D = A \div E \rightarrow I'_5 = \{E\}, O'_5 = \{D\} \\
T_6 : L = N \div E \rightarrow I'_6 = \{N, E\}, O'_6 = \{L\} \\
\hline
\end{array}
$$

There is only one order restriction:

$\sim \beta(T_2, T_5) \Rightarrow T_2 \perp T_5$(violates the output-independent condition)

T_1 and T_2 (or T_3 and T_4) can execute out of order now. Note that f(G) and g(H) can execute in parallel in this approach. By distinguishing between special and ordinary variables, the compiler can exploit more concurrency which the original Bernstein's conditions cannot detect.

4.2 Parallel Applications

Applications should be expressed without loss of their inherent parallelism. To extract maximum parallelism inherent in some parallel applications, programmers need to stateful computations.

4.2.1 Newnames

Suppose that an application needs to generate and use many different symbol names. In a regular single assignment functional language, it should be implemented by passing around parameters, n1, n2, n3, ..., as the program in Figure 4.1. The execution results are as follows: *values[1001, 1002, 1003, ...].*

Compared to using a state variable programming this way has the following three disadvantages:

1. The user has to use many variables. A for-loop is not sufficient to implement this application, because the new names may be needed in various places such as inside the branch of a conditional expression.

```
Program newname
function symbolname(aa: integer returns integer)
       1000+aa
end function
function newname(olda: integer return integer)
% One input parameter needed
       olda + 1
% olda is an ordinary variable
end function
function main(returns Integer)
 let a0:=0;
     a1:= newname(a0);
% This cannot be done by symbolname(newname(a0))
     name1:=symbolname(a1);
% a1 must be saved for passing to be used later
     a2:=newname(a1);
% Here a1 is used as an extra input-parameter, and
     name2:=symbolname(a2);
% a2 is an extra variable used for returned result
     a3:= newname(a2);   name3:=symbolname(a3);
...
   in name1,name2,name3,...
  end let
end function
end program
```

Figure 4.1: Newname program in regular Sisal

2. The function *symbolname* needs one extra input parameter and returns one extra result value.

3. It contains an artificially imposed serialization which limits the extraction of parallelism. The thread for expression *symbolname(a3)* cannot execute until all the other preceding invocations of *symbolname* have finished even if these threads are scheduled to different processing nodes. This is not a restriction inherent to the nature of the application itself since the application only requires that all new names be different from each other.

If a special variable is used to count the number of new names, then the above problems can be solved. See the example in Figure 7.2. One of the goals of functional languages is to reduce the artificially imposed serialization. This example demonstrates it is difficult for a pure single assignment language to achieve this goal. Our

work designed an extension to solve the above mentioned problems while keeping the advantages coming along with the nature of functional languages.

4.2.2 Parallel Branch and Bound Problem

The efficient parallel execution of the branch and bound application is made possible by an updatable global variable as shown in Figure 4.2. In this application, users try to find the shortest path from the starting node A to the goal node F. Assume

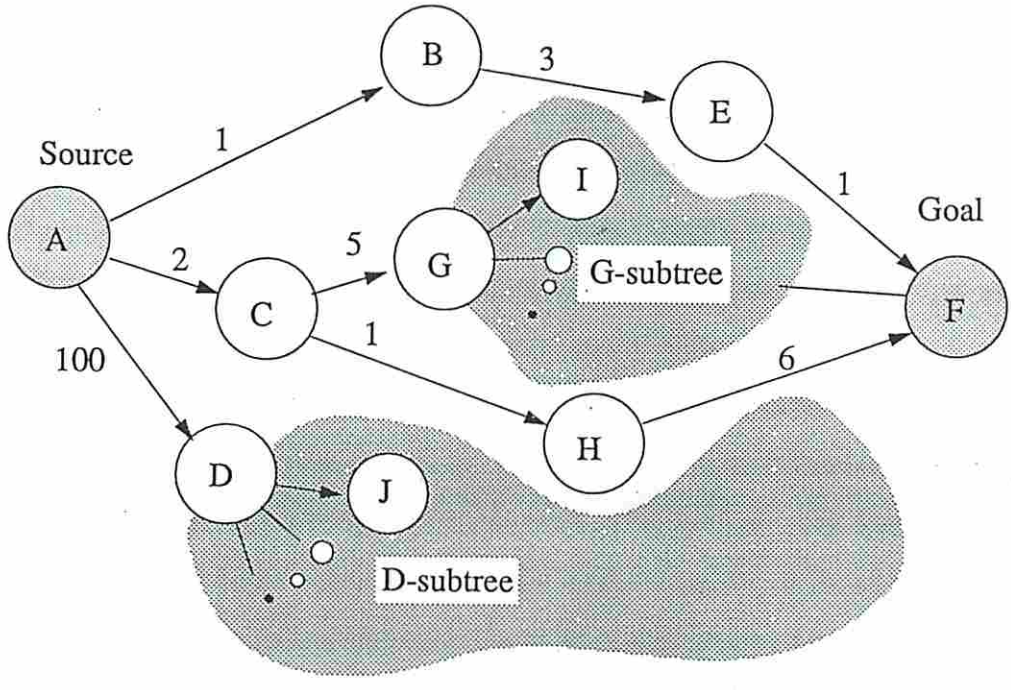

Figure 4.2: Parallel branch and bound algorithm

that in the beginning there are three parallel execution threads:

1. The first execution thread performs the calculation for the path: A $\rightarrow$ B $\rightarrow$ E $\rightarrow$ F.

2. The second execution thread performs the calculation for the path: A $\rightarrow$ C $\rightarrow$ ...

3. The third execution thread performs the calculation for the path: A $\rightarrow$ D $\rightarrow$ ...

Let us take a look at the following scenarios: Suppose that the first execution thread finishes, the value of the global minimum variable GL is assigned 5. Then the second thread performs the calculation until it reaches node G and the cumulated path length is 7, which already exceeds the value of GL. So the whole subtree rooted at G is pruned. The correct result is still guaranteed even if the whole G-subtree has been cut. When D is visited by the third thread, the calculated value is 100, which is greater than the current GL value, and so the subtree generated from D is pruned. On the other hand, when H is visited by the sub-thread generated from the second thread, the calculated value is 3, hence the subtree is kept alive. When F is reached through H, the calculated result is 4, which is less than the current value of GL, so GL is reassigned 4. If there are other threads alive, the calculation values of them should be compared with the current value of GL.

This is an algorithm with nondeterministic intermediate results and a deterministic final result. The use of the global variable GL, which is not allowed in a regular single assignment language, is the key to the easy programming and parallelism extraction for this application.

When a programming language lacks a feature, one can add the feature by changing semantics of the language to allow new operations as long as this change is not harmful to the original intent of the language. It is found that purely functional language lacks semantics of stateful computations, which are important in the practical cases of the real world. To be practically useful in many applications, computer programs must interact with outside world, and programmers must describe this interaction when they write the codes. However, this is not so easy in the realm of functional languages which totally prohibit imperative features since the real world is basically a global state which is waiting for programs to update it.

Chapter 5

Our Methodology

It is believed that adding a certain discipline of programming will help programmers to write more efficiently parallel programs. Our methodology is as follows:

1. The user is allowed to declare special variables, which are updatable and are global under the domain of the function which declares them. The domain of a function includes the expressions inside the function itself and the expressions inside its invoked functions.

2. The user can also declare ordinary variables. The term "special variable" rather than "global variable" is used because these variables can be "locally global" and can be declared as local somewhere in a program. See the example in Figure 5.5, where the variable V is declared as ordinary and used as a pure local variable inside function F. Our work can identify the serialization for the program, and recognize the expressions which must be executed sequentially.

We can make an orderly world of assignments so that the problems of imperative programming can be avoided while retaining states and assignments. The imperative set of an expression contains special variables which cause the expression to lose referential transparency. Since the only imperative features in extended-Sisal result from special variables, if the imperative set of an expression is empty then the expression is referentially transparent and is called a *pure* expression. A pure expression can be evaluated in parallel with any other expression because there will

be no interference between it and any other expression. If the imperative set of an expression is not empty, then this expression is called an *impure* expression. Placing an impure expression in different positions in a program may return different results. Figure 5.1 illustrates declarations of this type of variables in the extended-Sisal. The iset (imperative set) of Main is empty, therefore Main is a pure expression.

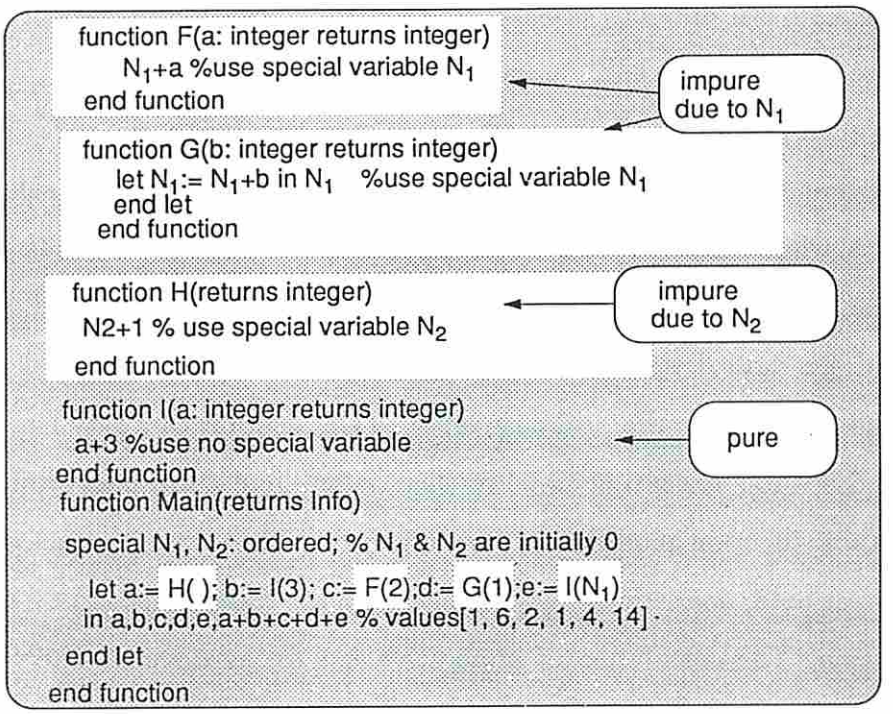

Figure 5.1: Declarations of special variables

The isets for these functions are as follows: iset(Main())=Φ, iset(F)=iset(G)=$\{N_1\}$, iset(H)=$\{N_2\}$, iset(I)=Φ. The isets for these expressions are as follows: iset(Main())=Φ, iset(F(2))=iset(G(1))=$\{N_1\}$, iset(H())=$\{N_2\}$, iset(I(N_2))=$\{N_1\}$. The imperative set of Main is empty, therefore Main is a pure expression. No matter where and how many times a pure function is used, it always returns the same answer for the same arguments.

5.1 The Scheme In Multi-threaded System

Functions which represent large amounts of work are significant sources for parallel execution. Arguments to a function can be independently evaluated. Low-level parallelism can be exploited by data-flow computers that overlap the execution of multiple functional units. In a multi-threaded system, if two threads have been scheduled to two different processors, then the execution of various parts of these two threads is ordered only by data availability.

For example, if threads T_1 and T_2 have been decided by the parallelism detection scheme that they can be executed out of order, and thread T_1 is scheduled to processor P_1 while thread T_2 is scheduled to processor P_2, then threads T_1 and T_2 can be executed in parallel. There may be an ordinary variable assigned by T_1 and read by T_2. This data dependency is easily taken care of by a "present" tag associated with the location for the ordinary value. This location can only contain one value during the whole execution time, so there is no nondeterminacy problem. There is no mutable location modified by T_1 and read by T_2 or vice versa. For an ordered special variable, nondeterminacy is solved by the order imposed by the imperative set scheme. The calculation of imperative sets for expressions and the parallelism identification based on the imperative isets are described as follows: There are two important definitions which will be frequently used in this chapter:

- **iset**: The iset (**imperative set**) of an expression is a set composed of global variables which are used in the expression of a program. These global variables cause the expression to lose *referential transparency*. An expression loses referential transparency if "putting it in different positions, we get different results."

- $\mathcal{U}$**iset**: The $\mathcal{U}$iset (**update_iset**) of an expression is a set composed of global variables which are updated in the expression. Since the set of *updated* global variables must be contained in the set of global variables, for any expression

$\mathcal{E}$, the following equation is always true:

$\mathcal{U}$iset($\mathcal{E}$) $\subseteq$ iset($\mathcal{E}$).

If the update_set of an expression is empty then this expression is non-destructive; that means the expression will not affect the value of any global variable. Otherwise, this expression is destructive; that means the expression will affect the value of at least one global variable. Two non-destructive expressions can be evaluated in parallel, even the intersection of the isets of these two expressions is not empty.

The problem of the expressive power limitation of functional languages can be solved by allowing the destructive updates of some particular variables. The methodology we propose is as follows:

1. the user is allowed to declare updatable global variables.

2. the imperative sets of the expressions in the program will be derived according to the declarations of the updatable global variables.

3. the expressions which can be executed in parallel with each other will be identified according to the imperative sets by performing some simple set of operations.

4. the expressions which must be executed in sequence will be recognized according to the imperative sets.

5. the data flow graph will be generated to indicate the parallelism according to the recognition of parallel and serial threads (the results of 3. and 4.).

For example in the program shown in Figure 5.2, the calculation of the isets for the expression e_4 is illustrated in Figure 5.3. The expression e_4 can be evaluated in parallel with any expression $\mathcal{E}$ if the intersection of iset($\mathcal{E}$) and iset(e_4) is empty. In this approach, users do not need to take the responsibility of isolating the imperative

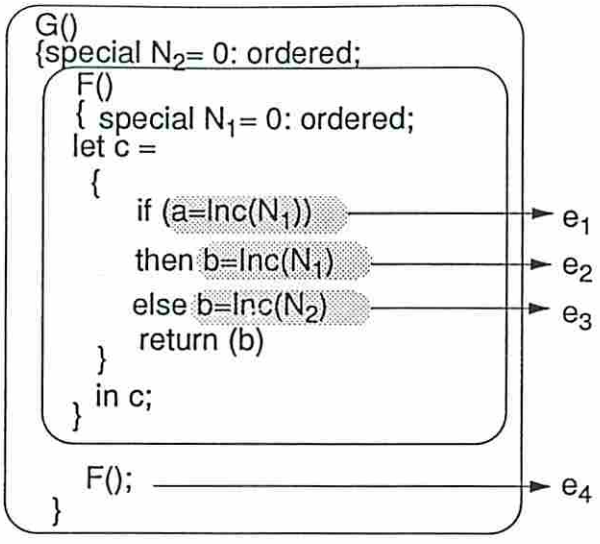

Figure 5.2: Impure expressions

Assume $e = f()\{e_1, e_2, \ldots e_n\}$

$$use(e) = \bigcup_{i=1,n} iset(e_i)$$

$$iset(e) = use(e) - dec(e)$$

For example:

$$use(e_4) = iset(e_1) \cup iset(e_2) \cup iset(e_3)$$
$$= \{N_1, N_2\}$$

$$dec(e_4) = \{N_1\}$$

$$iset(e_4) = use(e_4) - dec(e_4)$$
$$= \{N_2\}$$

Figure 5.3: The calculation for the iset

part from the remainder of the program. Instead, the compiler can manipulate it in an elegant and simple way. With this simple extension to Sisal 2.0, the execution environment should be able to easily identify the parallelism in programs. No matter where and how many times a pure function is used, it always returns the same answer for the same arguments. If the imperative set of an expression is not empty, then this expression is called an *impure* expression. Placing an impure expression in different positions in a program may return different results. The examples and uses of imperative sets are described in detail in [36]. The approach can extract more parallelism than simply applying Bernstein's conditions (Section 4.1). Relaxation algorithms such as the parallel branch and bound application are nondeterministic with regard to a global minimum variable. For example, in Figure 5.4, if the exe-

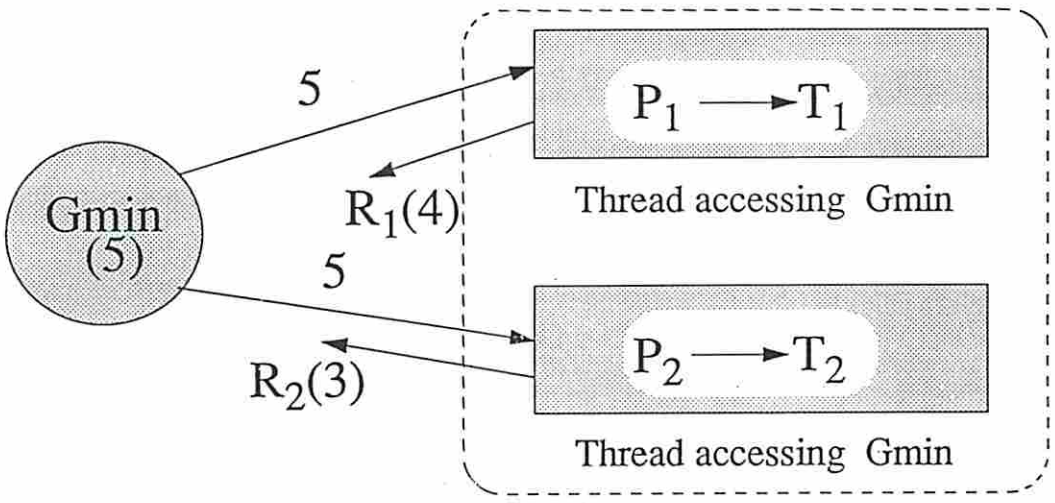

Figure 5.4: Threads in parallel branch and bound algorithm

cution of the thread T_2 completes earlier than T_1 and if its result (R_2) arrives the location of the global minimum (Gmin) earlier than T_1's result (R_1), then the results returned by T_2 will be overwritten and lost. If the result returned by T_2 is less than that returned by T_1, then the correctness of the execution cannot be guaranteed. This can be solved by adding the special function Monodec as follows:

```
function Monodec(GL2: integer returns integer)
```

```
% a critical section where GL is updated
                                    if GL <= GL2 then GL
                                    else let GL:= GL2;
                                         in GL
                                         end let
                                    end if
end function
```

The function Monodec can be invoked by only one thread at a time. The value of GL can be updated, and its value will be reduced each time it is updated. One example of programs is shown by using the Monodec function in Section 7. Note that a special variable never need to pass as a parameter while an ordinary variable is always passed by value.

5.2 Ordinary Variables

Variables are by default ordinary variables. In some occasions, however, an ordinary variable used in one function may have the same name as a special variable inside another function. If the former function is invoked by the latter function then this ordinary variable becomes a special variable. If the variable is not intended to be a special variable, the user may want to make sure that it will not incidentally become a special variable.

Our work allow users to declare a variable as ordinary inside a function, so that inside the function this variable will always be an ordinary variable (unless it is declared as special in the domain of the function). The ordinary variable declaration allows a style of programming where names for special variables can be reused as names for local variables in programs for unrelated computations. If a programmer declares an ordinary variable V inside a function F, then under the function F, V is always a local variable. V does not become a special variable even when the function F is invoked by a caller function where V is declared as a special variable. If F invokes a function G where V is declared to be special, then V is a special variable in the domain of G.

Sisal is not a "lazy" language and the physical arguments are evaluated before they are passed on to the called functions. Since passing an ordinary variable to be modified is meaningless, and since special variables declared by a caller function can be directly used in the invoked functions, they do not have to be passed through physical arguments. Thus, "call by value" is implemented in the extended-Sisal. The special functions FAA() and Monodec() are used to safeguard the correctness of the execution involving special variables updates by multiple threads executed in parallel. Special function FAA() and Monodec() will be described in Section 7.

```
program centaur6
function C(returns integer)
  c2 + c3;
% c2, c3 inherited as special variables when C() is called by A()
end let
end function
function A(returns integer)
special c2,c3 end special;
ordinary V end ordinary; %% V is declared as ordinary.
    let
       c1:=1;% This c1 is not related to the c1 in main()
             % A undeclared variable is by default an ordinary
             % variable
       c2:=2; c3:=3;
       V := 10;
    in V,C()+c1
   end let
end function
function main (returns integer)
special V end special; %% V is declared as special.
 let
   V := 1;
   c1 := 8; % c1 is ordinary and will not be modified by A()
   in A(),V,c1
 end let
end function
end program
```

Figure 5.5: Program using special variables

See the example in Figure 5.5, where the variable V inside the function main is a special variable, while the variable V inside the function A is an ordinary variable. Consequently, the assignment of V in the function A will not affect the value of the

special variable inside the function main. The execution results in this system is *values[10, 6, 1, 8]* instead of *values[10, 6, 10, 8]*. The variable c1 inside main and the variable c1 inside A() are two different ordinary variables by default, thus they do not interfere with each other's value.

The problem of the expressive power limitation of functional languages can be solved by allowing the update on some particular variables. Instead of being totally prohibited, declarations and updates on global variables are allowed. Programmers will be able to manipulate the stateful computation.

The semantics of extended-Sisal fit exactly between single assignment functional languages and the imperative languages in the language spectrum, and it will provide a higher programmability than both of them for many practical applications.

Chapter 6

Work With Centaur

The Centaur system [37] is used to construct a language environment. It has generated environments for ADA, FORTRAN, and LOTUS that feature parsers, structured editors, type checkers, and interpreters based on formal specifications. From the specifications of the syntax and semantics of a given language, one can automatically produce a syntactic editor and an interactive semantic tool for this language. Figure 6.1 shows the structure of the work with the Centaur system [36] [38]. Typol

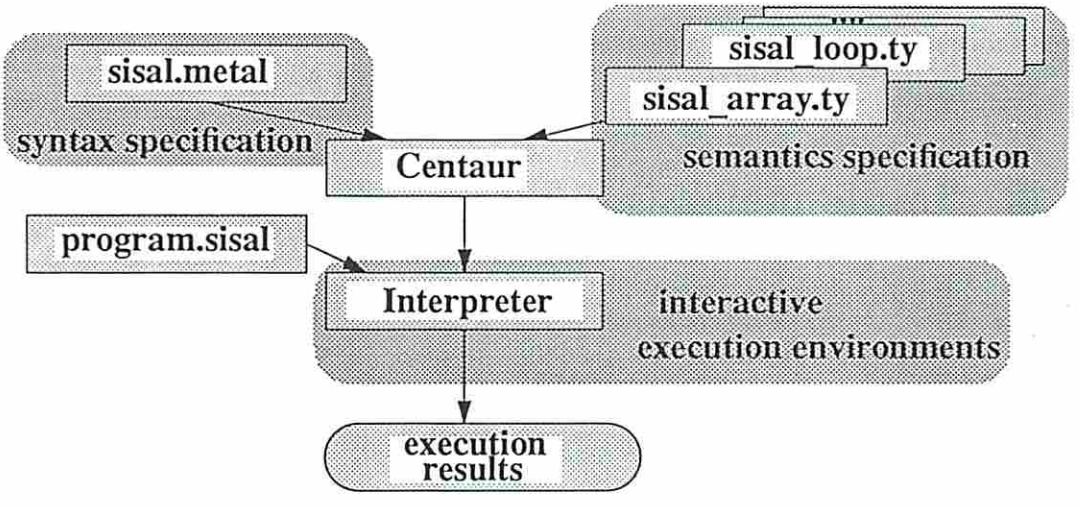

Figure 6.1: Work with the Centaur system

is a programming language within the Centaur system that implements natural semantics. Once A semantic specification for a language in Metal has been written,

users may execute the program in the Centaur system. Metal is a language used to specify the syntax of other languages. Writing a Metal specification for a language is the first step towards constructing the system. In Metal, the abstract syntax definition is independent of the concrete syntax of the language and the construction of the abstract syntax tree. Typol programs are independent of any interpreters or compilers that generate executable code. Semantic specifications are in an axiomatic style, using the Natural Semantics approach and its implementation uses the Typol formalism [39] [37]. An interactive programming environment for extended-Sisal 2.0 through its formal specifications is generated.

6.1 Working Environment

The Sisal environment implemented with Centaur tools is shown in Figure 6.2, where the program3 is being evaluated. Since the program is run in "debug mode", the "Examine" window can be used to show the values of the variables coming from the declaration part of the let, just before the function F called. Note that the "Value" window (window for execution result) is still empty. After the result is obtained, the environment in Centaur tool is shown in Figure 6.3.

6.2 Implement Sisal 2.0 Array in Centaur

Sisal 1.2 limits array construction to elements, rows, planes, and so on. Programmers requires more flexible array access [40]. Syntax in Sisal 2.0 [29] allows users to access matrices along arbitrary boundaries and construct matrices from components of arbitrary shape and size.

The design of Sisal 1.2 arrays is based on the one-dimensional array model. Under its semantics, arrays are constructed by the concatenation of the inner dimension arrays to build a single one-dimensional array. This model enables optimizations such as vectorization to be easily exploited because a multi-dimensional array is

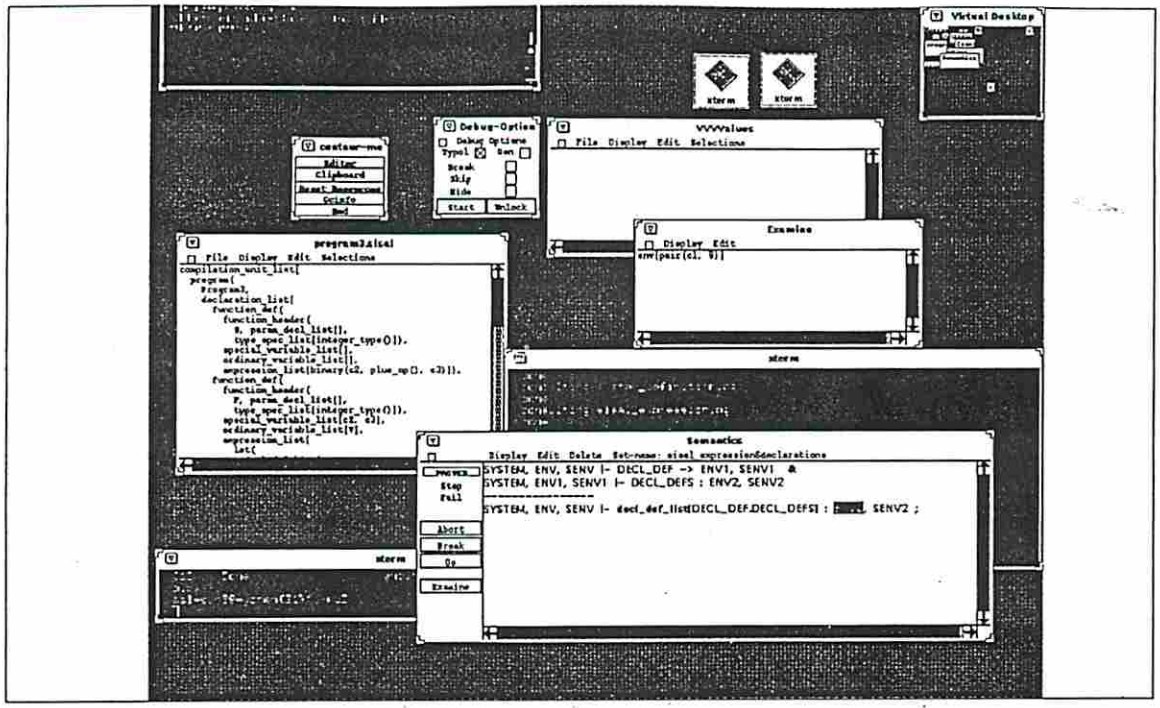

Figure 6.2: Sisal environment in Centaur tool in debug mode

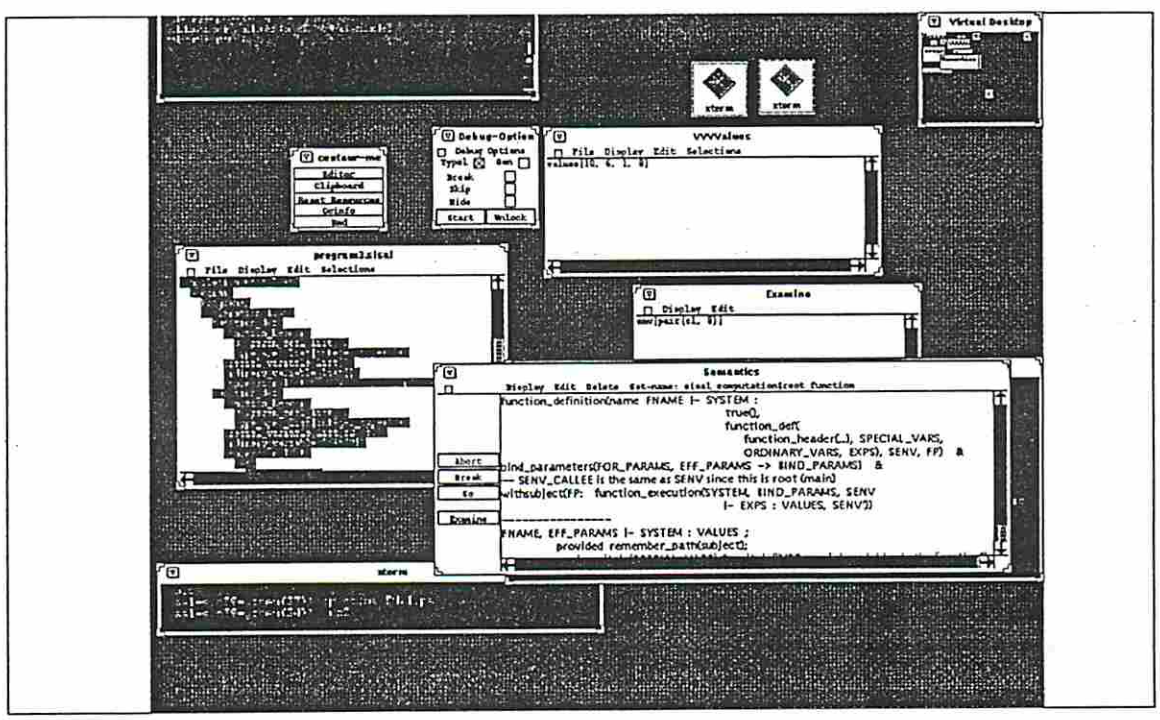

Figure 6.3: Execution result shown in the "Value" window

indeed represented as a vector containing vectors. Thus the construct of arrays is uniform. The disadvantages of this array design are (1) arrays must be stored in contiguous memory and (2) the inner vectors must always have the same size. For some applications, this is not flexible enough.

To fully utilize the advantages of one-dimensional array and avoid the disadvantages, the design of Sisal 2.0 array should allow both one-dimensional arrays and *real* multi-dimensional semantics.

6.2.1 Challenges for Semantic Specifications

The design of Sisal 2.0 arrays are intended to make the task of programming easier by allowing versatile syntax utilities. More convenience for programmers means more challenges in semantics specifications. Thus in this Centaur implementation, we have to write wise Typol rules to manipulate these versatile semantics of arrays.

We will show several array construct examples to illustrate the challenges as follows.

- ```
  B := array integer [7..8:30,40];
  C := array integer [7..8:50,60];
  A := array real [2..3, ..8:[2,..] B;
                             [3,..] C];
  ```

  In this case, the semantics must detect the size of B and C then put the elements of them into the appropriate position. Also the number 7 omitted in the index descriptor of array A must be detected by semantics rules.

- ```
  A := array real [2..3,7..8:[2,..] B;
                          [3,..] 0];
  ```

  The semantics coded in Typol rules must recognize the number '0' and put the appropriate number of '0' to the vector designated 3 in the array.

- ```
  A3 := array real [1..2,1..2,1..2,1..2:
                                    [1,1,1,1] 3.0; [1,1,1,2] 4.0;
                                    [1,1,2,1] 5.0; [1,1,2,2] 6.0;
  ```
  ```

```
                                    [1,2,1,2] 12.12;
                                    [2,2,1,2] 22.12;
                                    [2,2,2,2] 7.0;
                                    [otherwise] 3.1416 ];
```

The Typol rules must avoid overwriting the positions which are filled with the available values when inserting the value (3.1416) specified in "otherwise". In this implementation, the Typol rules fill the array construct with the otherwise value first then overwrite the positions which are specified with values (e.g. 3.0) in the array.

- ```
  let A15 := array integer[1,4,2];
  in
   array real [1..3,1..4: [1,A15] 11.1, 14.4, 12.2;
                          [2,A15] 21.1, 24.4, 22.2;
                          [otherwise] 99.9],
  ```

  In this array reference, array index can be specified by a predefined array, e.g. A15. The Typol rules must recognize that the index is an array name and then fetch the element from that array (A15) to feed into the index-descriptor to serve as position indexes.

- ```
      A2 := array real [2..3,7..8:[2,..] array real [1.0, 2.0];
                                  [3,..] array real [3.0, 4.0]];
  ```

  In this example, an inner array is created inside the array A2. Typol rules must identify that this is an array creation instead of one single element, so that the inner array elements will be inserted into the correct positions (there may be more than one positions) in the array A2.

6.2.2 Array Generation, Reference and Update

This section discusses in detail three important facilities of arrays in Sisal 2.0: array generation, reference to, and update of sub-arrays. We specify the semantics of them with Typol inference rules in the Centaur system. In program *sisal_expression*, three

rules recognize the expressions for *array generation*, *array reference*, and *array update* according to the operators array_gen, stream_ref_or_array_ref, and array_gen:

```
array_generate(SYSTEM,ENV,TYPE_SPEC|-SIZE_DESCR_LIST,
                                        ARRAY_PART_LIST->VALUES)
-------------------------------------------------------------------
SYSTEM, ENV |- array_gen(TYPE, SIZE_DESCR_LIST, ARRAY_PART_LIST)
                                        :  VALUES;

array_reference(SYSTEM, ENV |- ArrayName, ReferIndex -> VALUES)
-------------------------------------------------------------------
SYSTEM, ENV |- stream_ref_or_array_ref(ArrayName, ReferIndex)
                                      :  VALUES;

array_update( SYSTEM, ENV |- EXPRESSION, UPDATE_PART_LIST -> VALUES)
-------------------------------------------------------------------
SYSTEM, ENV |- array_update(EXPRESSION, UPDATE_PART_LIST) :  VALUES;
```

The semantics of array generation, array reference and array_generate are described as follows.

(1) The semantics of array generation can be written as follows:

```
build_array(SYSTEM,ENV,TYPE_SPEC|-SIZE_DESCR_LIST,ARRAY_PARTS
                                    ->DEF_ARRAY,DIM) &
fill_array(SYSTEM,ENV,DEF_ARRAY|-ARRAY_PARTS->ARRAY')
-------------------------------------------------------------------
SYSTEM,ENV,TYPE_SPEC|- SIZE_DESCR_LIST, ARRAY_PARTS-> values[ARRAY'];
```

First, we build the array with set *build_array*, specified as:

```
judgement SYSTEM,ENV,TYPE_SPEC|-SIZE_DESCR_LIST,ARRAY_PART_LIST
                                    ->VALUE,integer;

  otherwise(SYSTEM, ENV,TYPE_SPEC|- ARRAY_PART_LIST:VALUE)
 & build(SYSTEM,ENV,0,VALUE|- SIZE_DESCR_LIST -> ARRAY,DIM)
-------------------------------------------------------------------
  SYSTEM, ENV, TYPE_SPEC |- SIZE_DESCR_LIST,ARRAY_PART_LIST
                                  -> ARRAY,DIM;
```

The first premise (*otherwise*) returns a default value specified in the otherwise placement. Set *fill_array* examines each array_part in sequence. It is specified as follows:

```
set fill_array is
judgement SYSTEM,ENV,integer,VALUE|- ARRAY_PART_LIST->VALUE;

-- terminal
modify_array_with_array_part(SYSTEM,ENV,ARRAY_PART_COUNT,ARRAY
                |- ARRAY_PART->ARRAY')
-------------------------------------------------------------------
SYSTEM,ENV,ARRAY_PART_COUNT,ARRAY|- array_part_list[ARRAY_PART]
                -> ARRAY';

 modify_array_with_array_part(SYSTEM,ENV,ARRAY_PART_COUNT,ARRAY
                |-ARRAY_PART->ARRAY')
& plus1(ARRAY_PART_COUNT,ARRAY_PART_COUNT')
& SYSTEM,ENV,ARRAY_PART_COUNT',ARRAY'|- ARRAY_PARTS->ARRAY''
-------------------------------------------------------------------
SYSTEM,ENV,ARRAY_PART_COUNT,ARRAY
                |- array_part_list[ARRAY_PART.ARRAY_PARTS]
                -> ARRAY'';
end fill_array;
```

The first rule is fired when array_part_list has only one element and the recursive stops that way. Since set *fill_array* separately examines each array_part in sequence, we have to record that the processed array part. Set modify_array_with_array_part examine one ARRAY_PART and returns the filled array. The set is specified as:

```
judgement SYSTEM,ENV,integer,VALUE|- ARRAY_PART->VALUE;

 modify_array_with_opt_placement(SYSTEM,ENV,APC
                 |- OPT_PLACEMENT,EXPRESSION_LIST,ARRAY->ARRAY')
 ---------------------------------------------------------------
 SYSTEM,ENV,APC,ARRAY|- array_part(OPT_PLACEMENT,EXPRESSION_LIST)
                ->ARRAY';
```

In this rule, set modify_array_with_opt_placement examines the *placement* son (OPT_PLACEMENT) and decides whether it is a no_placement().

(A) If the *placement* son is no_placement(), then the *expression_list* son is examined by set modify_array_with_no_placement, specified as:

```
judgement SYSTEM,ENV,integer|- EXPRESSION_LIST,VALUE->VALUE;

  elt_to_elts(|- ARRAY->D,L,U,ELTS)
& modify_elts_by_exp_list(SYSTEM,ENV,APC,L |-EXPRESSION_LIST,ELTS
                          -> ELTS')
& elts_to_elt(D,L,U,ELTS'->ARRAY')
-----------------------------------------------------
 SYSTEM,ENV,APC|- EXPRESSION_LIST,ARRAY->ARRAY';
```

Set elt_to_elts is used to transform an array into its four feature values: dimension, lower, upper, and elements; set elts_to_elt is used to construct an array from four these array feature values.

The values of dimension, lower, upper, and elements are extracted from the initial array (ARRAY); ELTS is modified by the values contained in the EXPRESSION_LIST. Then the dimension, lower, upper and the modified ELTS together construct the new array (ARRAY') after processor the ARRAY_PART.

(B) If OPT_PLACEMENT is a PLACEMENT with contents, then the EXPRESSION_LIST is handled by set *modify_array_with_placement*, specified as:

```
extract_pattern_values(SYSTEM,ENV|- PLACEMENT,EXPS
                                -> PLIST,PATTERN,VALUES,ELTS)
& update(ELTS|-PLIST,PATTERN,VALUES,ARRAY->ARRAY')
-------------------------------------------------------------
SYSTEM,ENV|- PLACEMENT,EXPS,ARRAY-> ARRAY';
```

Set extract_pattern_values identify the pattern of the array part by the selector_part_list inside the PLACEMENT and get the values from the expression. Then, set *update*, called with these values, returns the final array.

(2) The semantics of *array reference* can be written as follows:

```
search_in_env(ArrName |- ENV: Array) &
eval_expression(SYSTEM, ENV |- Ref1_EXP : values[int_const Ref1])
ExtractArrElementsBy1Ref(Ref1 |- Array -> arrayElement)
```

```
-------------------------------------------------------
 SYSTEM,ENV|- ArrName, Ref1_EXP -> values[arrayElement];

 search_in_env(ArrName |- ENV: Array) &
 eval_expression(SYSTEM, ENV |- LOWER_EXP : values[int_const Ref1])
& eval_expression(SYSTEM, ENV |- UPPER_EXP : values[int_const Ref2])
& ExtractArrElementsBy2Ref(Ref1,Ref2|- Array -> PartialArr)
-------------------------------------------------------
SYSTEM,ENV|- ArrName, triplet(LOWER_EXP,UPPER_EXP, NE)
                              -> values[PartialArr];

 search_in_env(ArrName |- ENV: Array) &
 search_in_env(IndexArrName|-ENV: IndexArray) &
 getIndexELTS(|-IndexArray->IndexELTS) &
 modifyArrByIndexELTS(IndexELTS |- Array -> PartialArr)
-------------------------------------------------------
 SYSTEM,ENV|- ArrName, IndexArrName -> values[PartialArr];
```

Arrays can be referenced by an index integer(A1[3]), by a index triple (A1[3..4]), or by an index integer array (A[U]). The type are distinguished by the above 3 rules.

1. For type 1 expression, set ExtractArrElementsBy1Ref (called in the first rule,) called with the index, returns the referred array part.

2. For type 2 expression: set ExtractArrElementsBy2Ref (called in the second rule,) called with the lower and upper bound extracted from the index triple, returns the referred array part.

3. For type 3 expression: set modifyArrByIndexELTS (called in the third rule,) called with index array, returns the referred array part.

(3) The semantics of array update can be written as follows:

```
  search_in_env(NAME |-ENV: ARRAY)
& update_array_by_part_list(SYSTEM,ENV|-ARRAY,UPDATE_PART_LIST->ARRAY')
-------------------------------------------------------------------
SYSTEM,ENV|- NAME, UPDATE_PART_LIST -> values[ARRAY'];
```

First, according to the name of array, the content of the array (ARRAY) is extracted. This is done by the set search_in_env. Then set update_array_by_part_list, is called with ARRAY and UPDATE_PART_LIST.

Each update part in the UPDATE_PART_LIST is examined in sequence by set update_array_by_part_list. Called in set *update_this_array*, set *update_any_dim_array* examines one update part. This set defined as:

```
set update_any_dim_array is
judgement SYSTEM,ENV|- UPDATE_PART,VALUE->VALUE;

 eval_expression_list(SYSTEM, ENV |- EXPRESSION_LIST ->  VALUES)
 & process_selector(SYSTEM,ENV |- SELECTOR, VALUES, ARRAY -> ARRAY')
------------------------------------------------------------------
 SYSTEM,ENV|- update_part(SELECTOR,EXPRESSION_LIST),ARRAY->ARRAY';
end update_any_dim_array;
```

From the *expression_list* son, a list of values is generated. Then set *process_selector*, called with these values and the *selector* son, returns the updated array.

```
set process_selector is
judgement  SYSTEM,ENV |- SELECTOR, VALUES, VALUE -> VALUE;

classify_selector_part_list(SYSTEM,ENV
                          |-SELECTOR_PART_LIST,VALUES,ARRAY->ARRAY')
------------------------------------------------------------------
SYSTEM,ENV|-selector(SELECTOR_PART_LIST,diag_spec_list[]),
                          VALUES,ARRAY->ARRAY';
end process_selector;
```

(4) The semantics of arrays in loops can be written as follows:

```
set handle_array_of is
judgement SYSTEM,ENV,VALUE |-SIZE_DESCR_LIST,EXPRESSION,
                        FILTER_OR_AT_LIST:VALUE;

SYSTEM,ENV,OLD_VAL |- size_descr_list[],_,no_expression(): OLD_VAL;

remake_array_with_size_descr_list(ENV,OLD_ARRAY|-SIZE_DESCR_LIST
```

```
                        ->NEW_ARRAY)
-------------------------------------------------------------------
SYSTEM,ENV,OLD_ARRAY|-SIZE_DESCR_LIST,_,no_expression():NEW_ARRAY;
end handle_array_of;
```

The first rule handles the case in which the size_descr_list is empty, while the second rule handles nonempty size_descr_list cases.

6.2.3 Implementation Issues

This section briefly describes the Sisal environment implementation within the Centaur system in this section.

In the following Typol inference rule, which is in the set `eval_expression`, the expressions A1 and A2 are evaluated and then passed to the set array_concat.

```
eval_expression(SYSTEM,ENV|-A1:values[VALUE1])&
type(VALUE1 -> array_type(_,_)) &
eval_expression(SYSTEM,ENV|-A2:values[VALUE2])&
type(VALUE2 -> array_type(_,_)) &
array_concat(SYSTEM,ENV|-A1,A2->VALUES)
-------------------------------------------
SYSTEM,ENV|-binary(A1,concat_op(),A2):VALUES;
```

Although the rule is correct, it demands many evaluations for the expressions A1 and A2, hence it will slow down the evaluation. The rule has been modified to reduce these expression evaluations as follows:

```
eval_expression(SYSTEM,ENV|-A1:values[VALUE1])&
type(VALUE1 -> array_type(_,_)) &
eval_expression(SYSTEM,ENV|-A2:values[VALUE2])&
type(VALUE2 -> array_type(_,_)) &
array_concat(SYSTEM,ENV|-VALUE1,VALUE2=>VALUES)
-------------------------------------------
SYSTEM,ENV|-binary(A1,concat_op(),A2):VALUES;
```

The style of this rule is less *functional* because it sends the evaluation results of the expressions instead of the original expressions to its child set, i.e. `array_concat`.

However, the execution will not saturate the MU-PROLOG [37] memory and is faster because fewer evaluations are needed. Running the program in Figure 6.4, which performs quicksort on 18 elements [41], will return the proper result shown in Figure 6.5.

Some modifications have been made on the syntax of Sisal 2.0 [29] so that the syntax of the language can accept special variable declarations. For example, it can evaluate the following program:

```
program SAP1
   function main
      special A,B: ordered;
      let
           A:=1; B:=2;
           A:=A+10; B:=B+20;
      in A+100,B+200;
      end let
   end function
end program
```

and return the results: values[111,222].

6.3 Implementation of Array Data Types

This section presents the implementation of Sisal's array data type and will show various examples of array. Examples for many arrays follows:

```
program  ManyArrays
function main( returns Integer)
let
    A6 := array real [];
    A5:= array integer [10, 20, 30, 40];
    A1:= array integer [1..4: 10, 20, 30, 40];
    A3:= array integer [1..4: [2] 20; [1] 10; [3] 30; [4] 40];
    A2 := array real [2..3,7..8:[2,..] array real [1.0, 2.0];
                                [3,..] array real [3.0, 4.0]];
    A4 := array real [2..3, ..2:[2,1] 1.0; [2,2] 2.0;
                                [3,1] 3.0; [3,2] 4.0];
```

```
program QuickSortExample
type Info = array of integer ;

function QuickSort (Data : Info returns Info)
   if size(Data) < 2 then Data
   else let
        Pivot := Data[liml(Data)] ;
        Low, Mid, High := for E in Data do
                  returns
                     array of E when E < Pivot,
                     array of E when E = Pivot,
                     array of E when E > Pivot
                  end for ;
       in
     QuickSort(Low) || Mid || (QuickSort(High))
        end let
   end if
end function
function main ( returns Info)
  let
   A1 := array integer [ 1..5:53,14,92,10,65];
   A2 := array integer [ 1..5 : [ 5..1..-1] A1];
   A3 := array integer [ 1..3, -1..1:
            [i in 1..3,j in -1..1 {i dot j}] 1;
            [otherwise ] 0 ];
    A4 := A3 [ 2, 0 : 100];
    A5 := A3 [ 1, -1..1..2: -100];
    A6 := A1 + A2 * 2;
   in
    QuickSort ( A1 || A2 || A5 [1,..] || A6)
   end let
end function
end program
```

Figure 6.4: A quicksort program in Sisal 2.0

```
values[
  array(1,
    dim_content(
      1, 18,
   elts[
        -100, -100, 0, 10, 10, 14,
 14, 34, 38, 53, 53, 65, 65,
        92, 92, 171, 183,276]))]
```

Figure 6.5: Evaluation results for the quicksort program

```
    B := array integer [7..8:30,40];
    C := array integer [7..8:50,60];
    A7 := array real [2..3,7..8:[2,..] B;
                                [3,..] C];
    B := array real [7..8: 70, 80];
    A8 := array real [2..3,7..8:[2,..] B;
                                [3,..] 0];
A9 := array real [2..3,7..8:[2,..] array real [1.0, 2.0];
                                [otherwise] 0.0 ]  ;
  A10 := array real [2..3,7..8:[2,..] array real [1.0, 2.0];
                               [otherwise] 12.58 ] ;
  A11:= array integer [1..4: 10, 20, 30, 40];
 A12:= array integer [1..4: 10, 20, 30, 40];
 A13:= array integer [1..3,2..4: [1,..] 1, 2, 3;
                               [2,..] 4, 5, 6;
                               [3,..] 7, 8, 9];
 A14:= array integer [1..3,2..4: [1,..] 1, 2, 3;
                                  [2,..] 4, 5, 6;
                                  [3,..] 7, 8, 9];
 A15 := array integer[1,4,2];
 A16 := array real [1..4:1.0,2.0,3.0,4.0];
 U := array integer[1,4,2,1];
  A18 := array real [1..2,1..2,1..2:
                                    [1,1,1] 3.0; [1,1,2] 4.0;
                                    [1,2,1] 3.0; [1,2,2] 4.0;
                                    [otherwise] 0.0 ];
  A19 := array real [1..2,1..2,1..2,1..2:
```

```
                                        [1,1,1,1] 3.0; [1,1,1,2] 4.0;
                                        [1,1,2,1] 5.0; [1,1,2,2] 6.0;
                                        [1,2,1,2] 12.12;
                                        [2,2,1,2] 22.12;
                                        [2,2,2,2] 7.0;
                                        [otherwise] 0.0 ];
  A55:= array integer [2..3, 7..8: 10, 20; 30, 40];
in
A55,
 A16[U],
 array real [1..3,1..4: [1,A15] 11.1, 14.4, 12.2;
                        [2,A15] 21.1, 24.4, 22.2;
                        [otherwise] 99.9],
 A14[2,3..4],
 A13,
 A12[3..4],
 A18,
 A19,
 A1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11[3]
end let
end function
end program
```

Running the above program which performs many array data types will return the proper result shown as follows:

```
values[
  array(
    2,
    dim_content(
      2, 3,
      elts[dim_content(7, 8, elts[10, 20]),
                   dim_content(7, 8, elts[30, 40])])),
  array(1, dim_content(1, 4, elts[1.0, 4.0, 2.0, 1.0])),
  array(
    2,
    dim_content(
      1, 3,
      elts[
```

```
      dim_content(1, 4, elts[11.1, 12.2, 99.9, 14.4]),
      dim_content(1, 4, elts[21.1, 22.2, 99.9, 24.4]),
      dim_content(1, 4, elts[99.9, 99.9, 99.9, 99.9])])),
array(1, dim_content(1, 2, elts[5, 6])),
array(
  2,
  dim_content(
    1, 3,
    elts[
      dim_content(2, 4, elts[1, 2, 3]),
                  dim_content(2, 4, elts[4, 5, 6]),
      dim_content(2, 4, elts[7, 8, 9])])),
array(1, dim_content(1, 2, elts[30, 40])),
array(
  3,
  dim_content(
    1, 2,
    elts[
      dim_content(
        1, 2,
        elts[
          dim_content(1, 2, elts[3.0, 4.0]),
                      dim_content(1, 2, elts[3.0, 4.0])]),
      dim_content(
        1, 2,
        elts[
          dim_content(1, 2, elts[0.0, 0.0]),
                      dim_content(1, 2, elts[0.0, 0.0])])])),
array(
  4,
  dim_content(
    1, 2,
    elts[
      dim_content(
        1, 2,
        elts[
          dim_content(
            1, 2,
            elts[
              dim_content(1, 2, elts[3.0, 4.0]),
              dim_content(1, 2, elts[5.0, 6.0])]),
```

```
          dim_content(
            1, 2,
            elts[
              dim_content(1, 2, elts[0.0, 12.12]),
              dim_content(1, 2, elts[0.0, 0.0])])]),
      dim_content(
        1, 2,
        elts[
          dim_content(
            1, 2,
            elts[
              dim_content(1, 2, elts[0.0, 0.0]),
              dim_content(1, 2, elts[0.0, 0.0])]),
          dim_content(
            1, 2,
            elts[
              dim_content(1, 2, elts[0.0, 22.12]),
              dim_content(1, 2, elts[0.0, 7.0])])])])),
array(1, dim_content(1, 4, elts[10, 20, 30, 40])),
array(
  2,
  dim_content(
    2, 3,
    elts[dim_content(7, 8, elts[1.0, 2.0]),
                  dim_content(7, 8, elts[3.0, 4.0])])),
array(1, dim_content(1, 4, elts[10, 20, 30, 40])),
array(
  2,
  dim_content(
    2, 3,
    elts[dim_content(1, 2, elts[1.0, 2.0]),
         dim_content(1, 2, elts[3.0, 4.0])])),
array(1, dim_content(1, 4, elts[10, 20, 30, 40])),
array(0, dim_content(1, 0, elts[])),
array(
  2,
  dim_content(
    2, 3,
    elts[dim_content(7, 8, elts[30, 40]),
            dim_content(7, 8, elts[50, 60])])),
array(
```

```
    2,
    dim_content(
      2, 3, elts[dim_content(7, 8, elts[70, 80]),
             dim_content(7, 8, elts[0, 0])])),
  array(
    2,
    dim_content(
      2, 3,
      elts[dim_content(7, 8, elts[1.0, 2.0]),
             dim_content(7, 8, elts[0.0, 0.0])])),
  array(
    2,
    dim_content(
      2, 3,
      elts[
        dim_content(7, 8, elts[1.0, 2.0]),
            dim_content(7, 8, elts[12.58, 12.58])])),
  30]
```

6.4 Implementation of Array Operations

By default, the predefined function "size" returns the total number of elements composing its parameter. An optional second parameter can restrict the dimension of inquiry. Predefined functions "limh" returns the upper bounds of a single dimension of an array. Examples follow:

```
program ArrayProcessExample
type Info = array of integer ;

function call_array_process (n:integer returns Info)
   let

    A := array integer [1..4: [2] 20; [1] 10; [3] 30; [4] 40];
    S := size(A);                                          % (1)
    L := limh(A);                                          % (2)
    B,SUM :=  for i in [1..2] dot j in [3..4] do
                 returns array of (i+j), sum(j)            % (3)
              end for;
```

```
    A1 := array integer [ 1..5 : 53, 14, 92, 10, 65 ] ;    % (4)
    A2 := array integer [ 1..5 : [ 5..1..-1] A1 ] ;        % (5)
    A3 := array integer [ 1..3, -1..1:
                          [ i in 1..3, j in -1..1 {i dot j} ] 1;
                          [ otherwise ] 0 ];               % (6)
    A4 := A3 [ 2, 0 : 100];                                % (7)
    A5 := A3 [ 1, -1..1..2: -100];                         % (8)
    A6 := A1 + A2 * 2 ;                                    % (9)
   in
    n,A,S,A1,A2,A3,A4,A5,A6,
    A1 || A2 || A5 [1,..] || A6                            % (10)
   end let
end function
function main(returns Info)
  let m:=2;
   in call_array_process(m)
  end let
end function
end program
```

In the above example of Sisal 2.0 program, "Array Predefined Function", "Array Returned By For", "Array Generation", "Array Update", "Array Infix Operator" and "Subarray Selection" have been demonstrated:

- expressions (1) and (2) demonstrate the *Array Predefined Function*,
- expression (3) demonstrate *Array Returned By For*,
- expressions (4), (5), and (6) show *Array Generation*, Sisal has two ways for generating arrays. The first is via the "for" expression, and The second is using the array generator.
- expressions (7), and (8) illustrate *Array Update*, Sisal's facilities for array updating are a subset of facilities for generation and subarray selection. The

result of an update is a new array differing in the specified index positions or subarrays. The updates yield new arrays and do not affect the value of A3.

- expression (9) demonstrates *Array Infix Operator*, The infix scalar arithmetic operators + also accept array operands. Both array operand values must be conformable. As shown in the expression "$A1 + A2 * 2$" one operand, 2, of a dyadic operator, $*$, may be a scalar and used with each array element in the other operand, A2.

- expression (10) demonstrates *Subarray selection*, The syntax for subarray selection is as follows:

array-ref ::= primary [selector]

In this example, A5[1..], The leading primary, A5, gives the array value being accessed. The subpart of the selector can be one the following four cases:

1. a simple integer expression
2. triplet
3. named triplet
4. vector subscript

The above mentioned semantics of the array reference are all implemented in the Centaur system with Typol rules contained in the rule set "array_generate".

6.5 Typol Rules for Special Variables

Examples of the rules which are implemented with regards to the semantics of special variables are described in this section.

```
set init_special_variable is
judgement (SPECIAL_VARIABLE_LIST->ENV);
```

```
(special_variable_list[]->env[]);

(SVL->SENV)
-------------------------------------------
(special_variable_list[S.SVL]->env[pair(S,int_const 0).SENV]);

end init_special_variable;
```

Two rules are contained in the set "init_special_variable" under the Typol program:
"system_definition.ty", These two rules is used to set the default value for the special variables. The first rule handles the empty special variable case. The second rule can be fired recursively. If there are more than one special variable, this rule will be fired more than one times. In handling a special variable list with 2 special variables, the second rule will be fired twice and then the first rule be fired once. The set *init_special_variable* is fired by the following rule, which is one of the rules inside set *function_definition*.

```
init_special_variable(SPECIAL_VAR_LIST->LSENV)
-------------------------------------------------------------
NAME |- function_def(function_header(NAME,P,T),
                SPECIAL_VAR_LIST,ORDINARY_VAR_LIST,EXP_LIST):
        true(),
        function_def(function_header(NAME,P,T),
                SPECIAL_VAR_LIST,ORDINARY_VAR_LIST,EXP_LIST),
        LSENV, s(subject, int 2);
```

This rule recognizes SPECIAL_VAR_LIST from the abstract syntax tree, which is built by Metal specifications. After the execution of rule *init_special_variable* this rule obtains the value of LSENV and will provide the value of LSENV to the rule which provokes it.

The following rule show how the relationship between calling function and called function are handled with regard to the domain of special environments, which is the set of special variables. Note that if ordinary variables are declared inside the

called function, then the special variable with the same name must be eliminated from special environments.

```
appendtree(CALLEE_ADD_SENV,CURRENT_SENV',COMBO_SENV) &
kill_special_variables(COMBO_SENV,ORDINARY_VARS->COMBO_SENV') &
function_execution(SYSTEM, BIND_PARAMS,COMBO_SENV'
                   |- EXP_LIST:VALUES',NEW_COMBO_SENV)
& substract_env(NEW_COMBO_SENV,CALLEE_ADD_SENV
                   -> COMBO_EXCLUDING_CALLEE_SPECIAL) &
update_CURRENT_by_NEW_COMBO(CURRENT_SENV',
              COMBO_EXCLUDING_CALLEE_SPECIAL
                   -> NEW_CURRENT_SENV)
-------------------------------------------------
SYSTEM,ENV,CURRENT_SENV
              |- invocation(NAME, EXP_LIST):VALUES',NEW_CURRENT_SENV ;
```

The meanings of the predicates in this rule are explained as follows:

- The first predicate specifies that if there are a declared special variable inside the called function, it must be added to COMBO_SENV. All the upper level special environments must be combined together. Suppose function F1 calls function F2, and function F2 calls function F3, then inside F3, the special environments of functions F1, F2 and F3 must be combined for the evaluation of the expressions inside F3.

- The second predicate specifies that if the ordinary variables are declared in the called function then the special variables with their names appear in the ordinary variable list must be eliminated from the special environments of the called function.

- The third predicate fires the execution of the body of the called function with the newly formed special environments.

- The forth predicate is subtle. It distinguishes the special variables declared by the called function from the special variables not being declared by it but

updated by it. This rule does not directly return the original special environments of caller function to the rule which invokes this rule. The reason is that although the number of special variables of the original special environments and the result special environments will be the same, the values of them may have been modified.

- The last predicate updates the special environments of the called function with the imperative effect resulting from the called function.

The called function's *local special variables* does not affect the caller function's special environments at all. See the Appendix A for the detail Typol specifications.

6.6 Metal Specifications for Special Variables

The Metal specifications are used to parse the syntax of extended-Sisal and to construct the abstract syntax tree. Examples follow:

```
<Function_Def> ::= "function" <Function_Header>
 <Opt_Special_variable_List>
 <Opt_Ordinary_variable_List>
 <expression_List>   "end" "function" ;
 function_def(<Function_Header>,
   <Opt_Special_variable_List>,
   <Opt_Ordinary_variable_List>,
       <Expression_List>)
```

This rule recognizes the special variables (if any), the ordinary variables (if any), and the function body expressions, then put them into the syntax subtree built by

this rule. The variables such as Opt_Special_variable_List are not atomic and need to be further specified by Metal specifications.

```
<Opt_Special_variable_List> ::= ;

    special_variable_list-list(())

<Opt_Special_variable_List> ::= "special"

    <Special_variable_List> "end" "special" ";"

       <Special_variable_List>

<Special_variable_List> ::= <Special_variable>;

    special _variable_list-list((<special_variable>))

<Special_variable_List> ::= <Special_variable_List>
                                  "," <Special_variable>;

    special_variable_list-post(<Special_variable_List>,
                                  <Special_variable>)

<Special_variable> ::= <Name> ;

       <Name>
```

The Opt_Special_variable_List can be empty, containing one special variable, or containing more than one special variables. These specifications are explained as follows:

1. The first specification recognizes an empty Opt_Special_variable_List and constructs an empty list in the abstract syntax tree.

2. The second specification recognizes a non-empty Opt_Special_variable_List and builds a list with at least one element.

3. The third specification recognizes a one special variable, and constructs a list with one element.

4. The forth specification contain a recursive definition of Special_variable_List and allow the recognition and building of a list with more than one elements.

5. The last specification indicates a special variable is a Name.

```
<Name> ::= %NAME ;
        name-atom (%NAME)
```

See the Appendix B for the detail Metal specifications.

Chapter 7

Applications

Effective and efficient use of parallel computers is feasible by exploiting the parallelism inherent in applications. To determine the effectiveness of the extended-Sisal implementation, the comparison between the regular Sisal and extended-Sisal programs in terms of programmability, parallelism and performance is performed in this chapter.

7.1 Newname Problem

The newname application generates different names, and each name can be used in different functions. The programs in both regular Sisal and extended-Sisal are shown in the following discussion.

7.1.1 Programs

The regular Sisal program must be implemented by passing around parameters as shown in Figure 7.1. The execution results are as follows: *values[1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010].*

Extract maximum parallelism inherent in the nature of the application is difficult for the regular Sisal program. The extended-Sisal program for newname is shown in Figure 7.2. The execution results can be one of the following 10! cases:
values[1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010],

```
program newname
function symbolname(aa: integer returns integer)
          1000+aa
end function
function newname(olda: integer returns integer)
% One input parameter needed
          olda + 1
% olda is an ordinary variable
end function
function main(returns Integer)
     let a0:=0; % time: 1 (assign)
          a1:= newname(a0);
% time: 2 (plus), 3 (assign)
% This cannot be done by symbolname(newname(a0))
          name1:=symbolname(a1);
% time: 4 (plus), 5 (assign)
% a1 must be saved for passing to be used later
          a2:=newname(a1);
% time: 6 (plus), 7 (assign)
% Here a1 is used as an extra input-parameter, and
% a2 is an extra variable used for returned result
          name2:=symbolname(a2);
% time: 8 (plus), 9 (assign)
          a3:= newname(a2);   name3:=symbolname(a3);
% time: 10, 11, 12, 13
          a4:= newname(a3);   name4:=symbolname(a4);
% time: 14, 15, 16, 17
          a5:= newname(a4);   name5:=symbolname(a5);
% time: 18, 19, 20, 21
          a6:= newname(a5);   name6:=symbolname(a6);
% time: 22, 23, 24, 25
          a7:= newname(a6);   name7:=symbolname(a7);
% time: 26, 27, 28, 29
          a8:= newname(a7);   name8:=symbolname(a8);
% time: 30, 31, 32, 34
          a9:= newname(a8);   name9:=symbolname(a9);
% time: 35, 36, 37, 38
          a10:= newname(a9);  name10:=symbolname(a10);
% time: 39, 40, 41, 42
     in name1,name2,name3,name4,name5,name6,name7,name8,name9,name10
% time:43
     end let
end function
end program
```

Figure 7.1: Newname program in regular Sisal

```
program newname          % This program is to generate different names
function symbolname(a: integer returns integer)
   1000+a                % time: 4
end function
function assignName(returns integer)
  symbolname(FAA(1)) % time: 3
                         % N is updatable global variable
end function
function main(returns Integer)
 special N end special;
                         % Initially, special variable N is 0. time:1
     assignName(),assignName(), assignName(), assignName(),
                         % No input parameter is needed.
     assignName(),assignName(), assignName(),assignName(),
     assignName(),assignName()
                         % time: 2, 16

% These ten threads can be executed concurrently
end function
% FAA special function, where N is a special variable
% FAA can only be invoked by one thread at a time
special function FAA(V: integer returns integer)
                          let N := N + V
% time: 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
                          in  N
                        % time: 15
                          end let
end function
end program
```

Figure 7.2: Newname program in extended-Sisal

values[1002, 1001, 1003, 1004, 1005, 1006, 1007, 1008, 1009, 1010],

..., and values[1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008, 1010, 1009].

Parallelism analysis for the extended-Sisal program is shown in Figure 7.3.

7.1.2 Performance Analysis

The execution time for regular Sisal program is 43 units, and the execution time for extended-Sisal program is only 15 units. The performance improvement, the ratio of execution time, of regular Sisal over extended-Sisal program is

$$\xi_{newname} = 43/15 = 2.866. \tag{7.1}$$

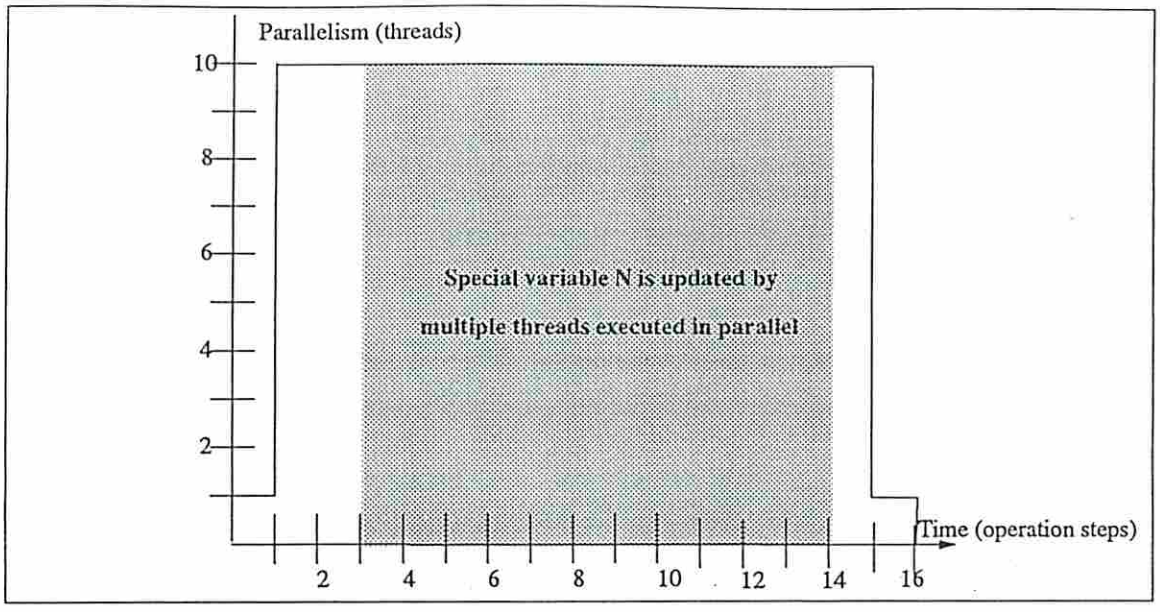

Figure 7.3: Parallelism analysis of newname program in extended-Sisal

Moreover, in this simple example, the names is directly returned to the results; if the names is used by some functions, then the ratio will be much larger. The trick is that the order of the names is insignificant, and extended-Sisal allow users to make use of this property to explore more parallelism. If these names are generated and used in various functions, which is practical, the performance improvement is much more significant. For example, if name1 is used by function F1, name2 is used by F2, in regular Sisal program, F2 cannot execute until F1 is finished and returns the name parameter to F2. The extended-Sisal program for newname with functions using names is shown in Figure 7.4. Suppose the execution time of Fi is T_{Fi}, where $i = 1, ..., 10$. For the execution time of a regular Sisal program is

$$T_{reg} = T_{F1} + T_{F2} + ... + T_{F10} + T_{Nreg}. \tag{7.2}$$

For the execution time of extended-Sisal program is

$$T_{ext} = maximum(T_{F1}, T_{F2}, ..., T_{F10}) + T_{Next}. \tag{7.3}$$

```
program newname
% generate 10 different names and use them in functions
% F1, F2, ..., F10.
...
function main(returns Integer)
 special N end special; % Initially, special variable N is 0. time:1
      in F1(name1),F2(name2),F3(name3),
         F4(name4),F5(name5),F6(name6),F7(name7),
         F8(name8),F9(name9),F10(name10)
     end let
end function
...
```

Figure 7.4: Newname program with functions using names in extended-Sisal

Where N_{reg} is the time for generate new names in regular Sisal, and T_{Next} is the time for generate new names in extended Sisal. The performance improvement is

$$\xi_{newname} = T_{reg}/T_{ext}. \tag{7.4}$$

Since $N_{reg} > T_{Next}$, if

$$T_{F1} + T_{F2} + ... + T_{F10} >> maximum(T_{F1}, TF2, ..., T_{F10}), \tag{7.5}$$

then

$$\xi_{newname} >> 1. \tag{7.6}$$

As derived in equation 7.1, $T_{Nreg}/T_{Next} = 2.87$. Assume $T_{F1} = T_{F2} = ... = T_{F10} = TF$, then

$$\xi_{newname} = T_{reg}/T_{ext} = (10 * TF + 2.87 * T_{Next})/(TF + T_{Next}). \tag{7.7}$$

Assume $\gamma = TF/T_{Next}$ and apply equation 7.2 and 7.3 to 7.7, the following equation is derived:

$$\xi_{newname} = (10 * \gamma + 2.87)/(\gamma + 1). \tag{7.8}$$

If the new names are directly returned, then $\gamma = 0$, otherwise $\gamma > 0$. The performance improvement in terms of γ is shown in Fig 7.5. The parallelism approaches

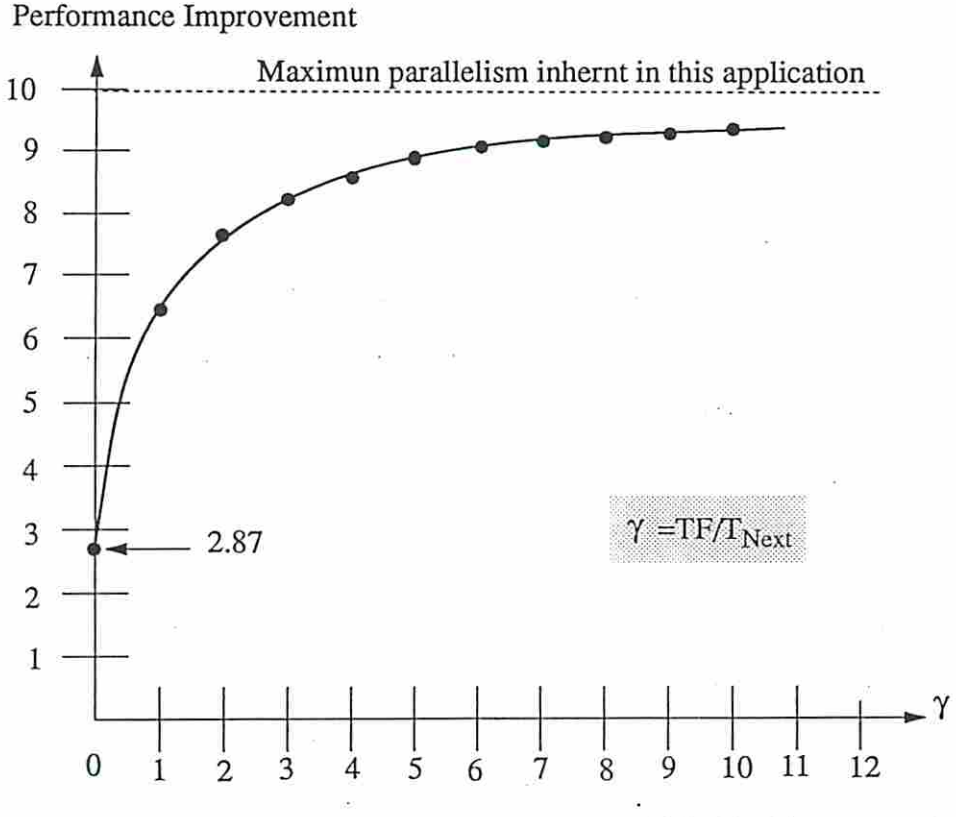

Figure 7.5: Performance improvement of newname problem with name using functions

10, which is the number of different names. If the number of different names are more, then the potential parallelism extracted are far more than the regular Sisal. This figure show the parallelism restriction with regard to restriction of the names generation has been relaxed by the extended-Sisal to its potential maximum parallelism inherent in the application. This is the best any parallel programming can extract from this particular application. The extended-Sisal programs are able to reach this only by using a single special variable.

7.1.3 Discussion

In the extended-Sisal program, the function *symbolname* does not need one extra input parameter or returns one extra result value. It reduces the artificially imposed serialization. The thread for expression *symbolname(a10)* can execute out of order or concurrently with all the other nine invocations of *symbolname* if they are scheduled to different processing nodes. Several aspects of programs are discussed:

- *programmability*: The extended-Sisal program is easier to code since no parameters need to be passed around.
- *efficient parallel processing*: The benefit of functional language– easy parallel processing is kept, because only the special variable N is of concern in terms of nondeterminacy, and it can be easily singled out and taken care of.
- *necessary determinacy*: FAA is used to protect the necessary determinacy by ensuring the mutual updates to N.
- *unnecessary determinacy*: The unnecessary determinacy is reduced. The new names can be assigned to the functions F1, F2, ..., F10 as follows:
  A combination of names, $N_1, N_2, ..., N_{10}$ *is an acceptable name assignment to (F1, F2, ..., F10)* $\Longleftrightarrow N_i \neq N_j$ *for all* $i \neq j$.
  The nature of the application does not specify which combination is the required one. Due to the expressive power limitation, the regular Sisal program is

unnecessarily "hardwired" to ensure exactly one combination, and artificially impose unnecessary determinacy.

- *No redundant concurrent execution*: The function *Symbolname* is used to customize a number to a symbol name such as Room1, Flight002, ..., etc. The longer it takes to execute *Symbolname*, the more the execution are overlapped. This kind of parallel execution are not redundant. The redundant execution and its reduction in shortest path problem will be discussed.

7.2 Histogramming Problem

The histogramming application counts the number of elements with a certain value in an array. In this example, the user needs to know the number of elements with value 0 in array A1, , which contains the data sets: *[0, 2, 0, 4, 0, 6, 7, 8 , 1, 1]*. The programs in both regular Sisal and extended-Sisal will be shown in the following section.

7.2.1 Programs

The regular Sisal program for histogramming is shown in Figure 7.6. Since the non-product form of for loop impose unnecessary ordering, it is difficult to extract maximum parallelism from the nature of the application if users implement it in regular Sisal. For counting the number of elements with certain value there is no need to determine in the program which element should be examined first, second, etc. This is a determinacy which not inherent in the application. All kinds of regular Sisal compilers will not parallelism the non-product form for loop, because they "know" there is some harmful nondeterminacy inside the non-product form for loop. They will be able to parallelize the product form for loop, and simply give up the non-product form. Parallel programmers may be frustrated by the fact that the application such as histogram contains parallelism but has to be implemented in

```
program histogram
type IntOneDim = array of integer ;
function histogram (N: Integer, digit: IntOneDim
                                          returns IntOneDim)
 For Initial
    Slot := Array_Fill(0,8,0);      % time: 3
    i:= 1;                          % time: 4
 while i < N repeat                 % time: 5,9,13,...,45
 slot := old slot[digit [old i]:    % time: 7,11,...,43 (assign)
                 old slot[digit[old i]]+1];
                                    % time: 6,10,...,42 (plus)
    i := old i +1;                  % time: 8,12,...,44
  returns value at slot             % time: 46
 end for
end function
function main(returns Integer)
 let A1 := array integer [1..10: 0, 2, 0, 4, 0, 6, 7, 8, 1, 1] ;
                                    % time: 1
     B1 := histogram(11,A1)         % time: 2 (invocation),
                                    % time: 47 (assignment)
  in B1[0]
                                    % time: 48
                                    % Only after array B1 is ready,
                                    % then B1[0] can be used.
 end let
end function
```

Figure 7.6: Histogramming program using regular Sisal

non-product form for loop and has to be artificially serialized. The reason is that the regular Sisal does not distinguish the necessary determinacy from the unnecessary determinacy. Extended-Sisal, however, can distinguish the necessary determinacy determinacy by implementing special user declared variable. This will be further discussed in Section 7.2.2. The extended-Sisal program for histogramming is shown in Figure 7.7. This is a product form for loop, the array elements can be fed into FAA in parallel. If the elements of the array are from different threads and are available in different time, this style of programming will not enforce a artificial ordering. Compared with the regular Sisal program of this application, more parallelism can b exploited. Parallelism analysis for the histogramming in extended-Sisal is shown

```
program histogramExample
type  Info = array of integer;
function histogram(Data: Info returns Info)
   for E in Data do                    % time: 3
            returns
             if E = 0 then             % time: 4
                  FAA(1)               % time:5,6,7
             else 1
            end if
         end for                       % time: 8
end function
function main(returns Integer, Integer)
special COUNT0 end special;
 let A1 := array integer [1..10: 0, 2, 0, 4, 0, 6, 7, 8, 1, 1] ;
                                       % time: 1
  in histogram(A1),                    % time: 2
     COUNT0                            % time: 9
 end let
end function
% FAA can only be invoked by one thread at a time
special function FAA(V: integer returns integer)
                        let COUNT0 := COUNT0 + V
                                       % time: 5, 6
                        in  COUNT0 % time: 7
                        end let
end function
```

Figure 7.7: Histogramming program using extended-Sisal

in 7.8. The execution results for both programs are *values[3].*

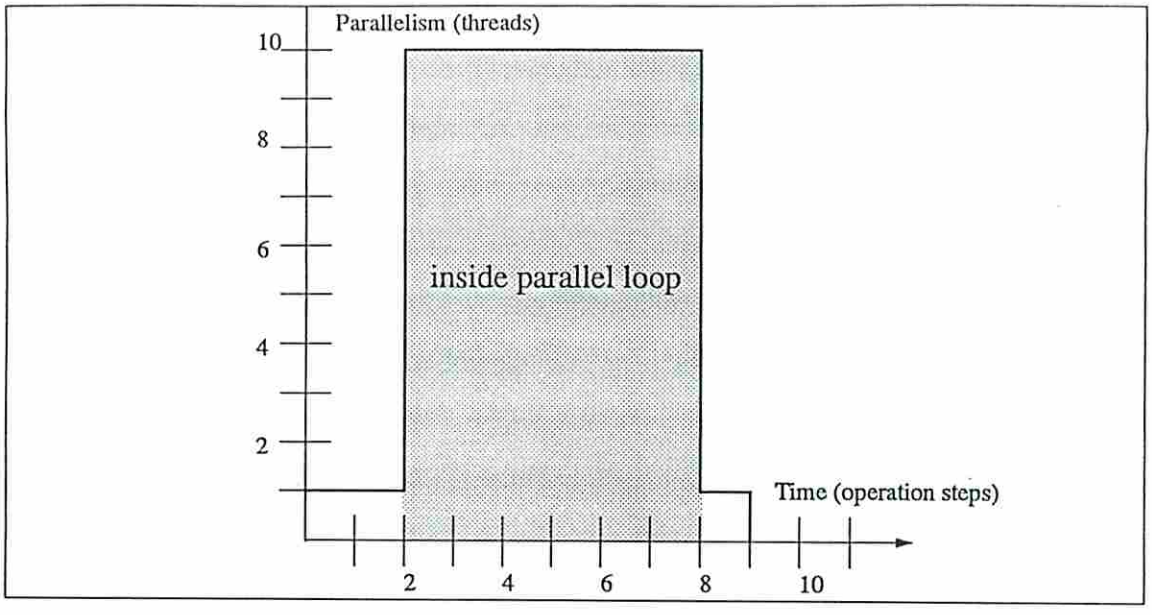

Figure 7.8: Parallelism analysis of histogramming program in extended-Sisal

7.2.2 Performance Analysis

The execution time for regular Sisal program is 48 units, and the execution time for extended-Sisal program is only 9 units. The performance improvement (execution time ratio) of regular Sisal over extended-Sisal program is

$$\xi_{histgram} = 48/9 = 5.33. \tag{7.9}$$

Moreover, in this example, the size of the target array is only 10; if the array size is bigger, then the ratio will be larger. The trick is: in this application, the order of updating the counter is not important, extended-Sisal allow users to make use of this property to explore more parallelism.

Suppose that the elements of the data set are generated and available at different times, these elements can be fed in parallel. This nature of the application is difficult to explore in a regular Sisal program, but can be easily implemented in extended-Sisal.

7.2.3 Take Advantage of Hardware I-structure Array

In a hardware I-structure array, individual array element can be available at different times. Suppose the second array element is not available yet, then the regular Sisal program will have to wait for its availability before it can proceed the calculation on the third element. The extended Sisal program can take advantage of the feature of an I-structure array and go on to compute on the third and other elements without having to wait for the availability of a specific element. In Figure 7.9, the histogramming function sends requests for data to the I-structure array A, which contains 100 elements. The requests are shown with dashed arrow lines. The array

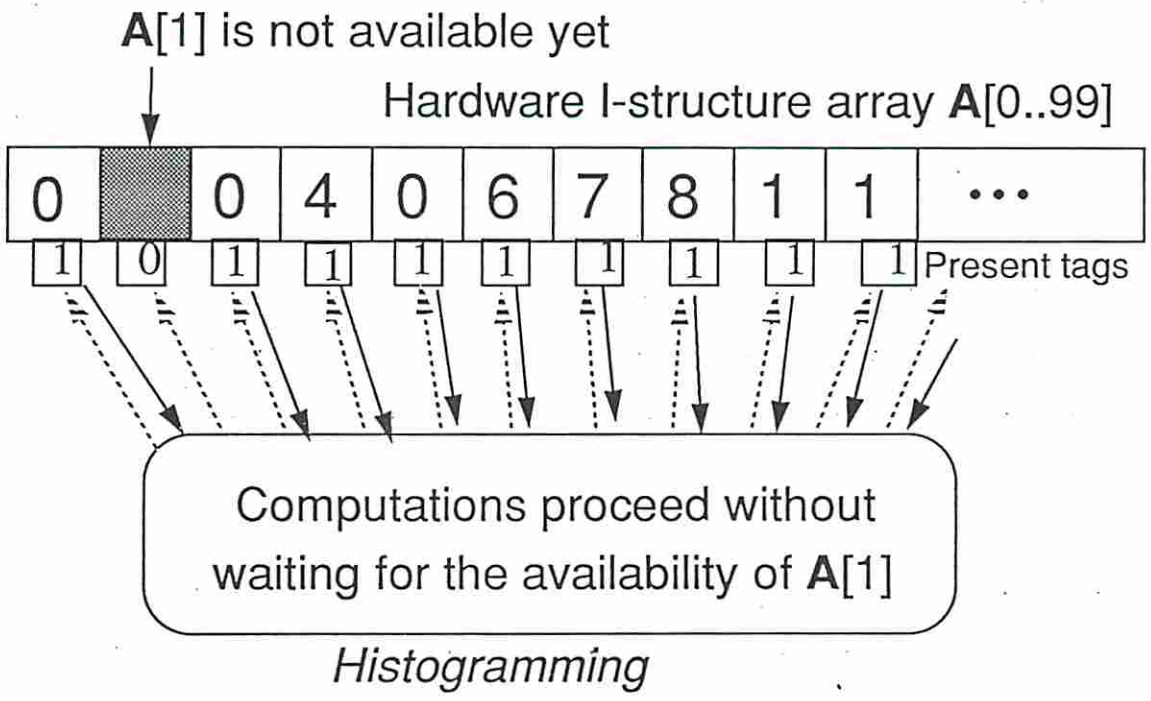

Figure 7.9: Extended-Sisal let histogramming take advantage of I-structures

A then returns any individual element which is already available to the histogramming function. The data movements are shown by solid arrow lines. The waiting time for the availability of A[1] is "overlapped" with the calculation on the other 99 elements. Whenever element A[1] is available, the execution of histogramming can

be completed in a short period of time. Indeed, the values of array elements may be derived upon the request of the histogramming function. The artificial ordering which would be imposed by a regular Sisal program is relaxed.

7.3 Shortest Paths Problem

In this problem, users are given a directed graph G = (V,E), a cost array COST(E) for the edges of G, a source node A and a goal node B. The problem is to determine the length of the shortest paths from A to B. It is assumed that all the costs are positive. This application is practical and useful because graphs may be used to represent the highway structure with vertices representing cities and edges representing sections of highway [42, 43]. The edges may then be assigned costs which might be the distance between the two cities connected by the edge. A motorist wishing to drive from city A to city B would be interested in answers to the question: If there is more than one path from A to B, which is the shortest path? The length of a path is defined to be the sum of the costs of the edges on that path. The starting node of the path will be referred to as the *source* and the last node the *goal.* The graphs will be digraphs to allow for one way streets (Figure 7.10).

7.3.1 Programs

In regular Sisal, the application cannot be programmed by a global variable (Figure 7.11.) The execution results is *values[15].* In a extended-Sisal program, it can be implemented by the global variable GL as shown by the program in Figure 7.12. The execution results is: *values[15,15].*

In the regular Sisal program: (1) Each thread has to collect all the returned values for each iteration of the loop. (2) Each thread has to decide the minimum value for these returned values. Finding a minimum is in O(N) time, where N is the number of elements in the target array. In the extended-Sisal program: (1) A

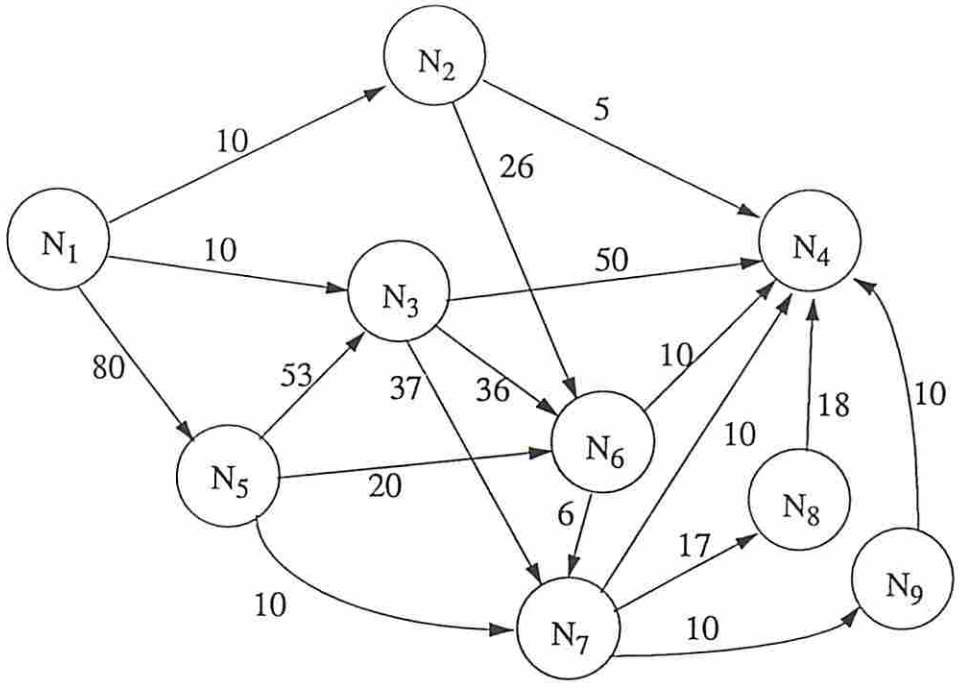

Figure 7.10: Graph for finding shortest path from N_1 to N_4

thread does not have to wait until the entire iteration is finished before it updates GL. (2) A thread does not have to collect all the returned values for each iteration. Collecting data into an array is not needed since the useful message of each iteration is stored in GL. (3) The decision of minimum value for the intermediate result for GL has been scattered into all iterations. Whenever there is a new value for GL, it is compared against the current GL. This almost eliminates the searching time and improves the performance, since the comparison is amortized and hidden in the parallel executed threads.

In this program, the declaration of the special variable GL is allowed. Regular Sisal programs do not manipulate storage cells for ordinary variables; the task of assigning values of ordinary variables to memory locations is actually left to the compiler. The special variable scheme returns control of memory location assignment

```
program BB
function V(Data:Info, L1: integer, C: integer returns integer)
 if C = 4 then L1 % reach the goal node
   else
     let A1:= for N in array integer [2..9: 2, 3, 4, 5, 6, 7, 8, 9]
              do L2:= L1+Data[C,N]
                 returns
                 array of V(Data,L2,N) % need time for the
                                       % collection of an array
              end for
    in
    minval(A1)     % a reduction returns the minimum element of A1
    end let
 end if
end function
function main(returns integer)
special GL end special;
let
  COST:= 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, 8] 10;
              [8, 1] 18; [9,4] 10;
              [otherwise] 9999 ];
in  V(COST,0,1)
end let
end function
end program
```

Figure 7.11: Shortest path application programmed in regular Sisal

```
program BB
function V(Data:Info, L1: integer, C: integer returns integer)
 if C = 4 then L1 % reach the goal node
 else for N in array integer [2..9: 2, 3, 4, 5, 6, 7, 8, 9]
           do  L2:= L1+Data[C,N]
               returns
               if GL <= L2 then GL % cut off the subtree
               else   let GL2:= V(Data,L2,N) % invoke a thread
                      in Monodec(GL2)
                      end let
              end if
     end for
 end if
end function
% a Monodec special function, where GL is special
% Monodec can only be invoked by a thread at a time
special function Monodec(GL2: integer, returns integer)
                                 if GL <= GL2 then GL
                                 else let GL:= GL2;
                                      in GL
                                      end let
                                 end if
end function
function main(returns integer)
special GL end special; %a special variable denoting global length
let GL:= 9999; % 9999 represents infinity
    COST:= 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; [8,1] 18; [9,4] 10;
                  [otherwise] 9999 ];%denotes no connection
   in  V(COST,0,1)
end let
end function
end program
```

Figure 7.12: Shortest path application programmed in extended-Sisal

to programmers. This program will produce exactly the same final results on a single workstation or a multicomputer.

7.3.2 Performance Analysis

The use of the global variable GL is the key to the easy program and parallelism extraction. Regular Sisal cannot express the nondeterministic intermediate results and therefore cannot exploit the parallelism which extended-Sisal can exploit.

The parallelism for the shortest path program can be derived by analyzing the execution tree. Figure 7.13 illustrates an execution tree for the program in regular Sisal 2.0. Figure 7.14 illustrates an execution tree for the program in extended-

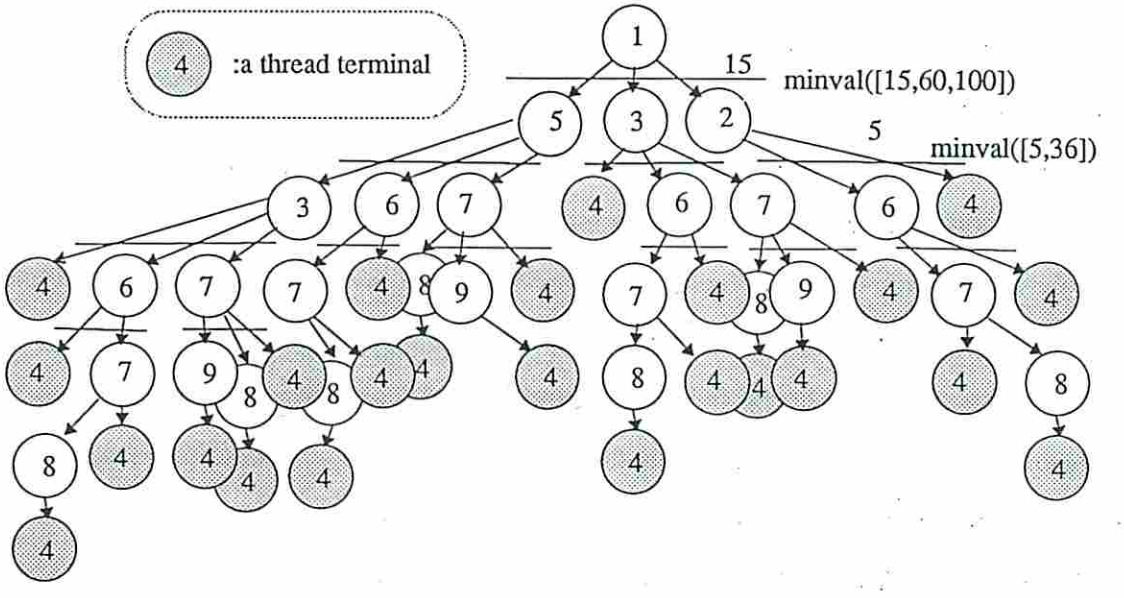

Figure 7.13: Execution tree for shortest path program in regular Sisal

Sisal 2.0. The degree of parallelism for Figure 7.13 is greater than that of Figure 7.14. However, the execution tree in Figure 7.14 contains almost only useful operations, while the tree in Figure 7.13 includes many nonessential operations. These nonessential operations appear simply because the expressive power of regular Sisal is not sufficient to efficiently reduce them. Indeed, the execution time is the real

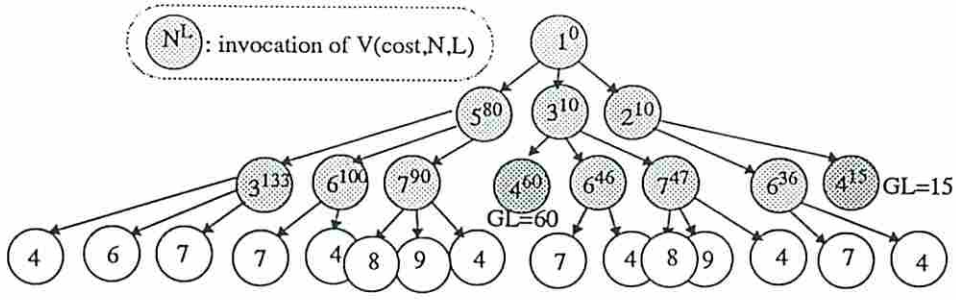

Figure 7.14: Execution tree for shortest path program in extended-Sisal

concern for users. Compared to the program written in extended-Sisal, the regular Sisal program is executed with more operations because the cuts of subtrees cannot be implemented because of the lack of a global updatable variable GL. In addition, its execution also occupies more resources (storage locations, processors) than the extended-Sisal program since there are more threads executed at a time. Figure 7.15 illustrates the degree of parallelism and the nodes visited in each level for both of the regular Sisal and extended-Sisal programs. Figure 7.16 illustrates the number of invocations and the estimated execution time for both extended-Sisal and regular Sisal programs. In this figure, constant k_1 denotes the initial array assignment time plus the final result returning time. Constant k_2 denotes the time spent in one level invocation for regular Sisal program. Constant k_3 is the time spent in one level invocation for extended-Sisal program. The ratio k_2/k_3 is a constant and is not related to data size, i.e., the input array size. T is the minimum element finding time in regular Sisal program. Extended-Sisal programs do not need this time since the comparison of minimum value is amortized and overlapped by using a GL among threads. T is in O(log(N)) time. L_1 is the number of execution levels needed in the regular Sisal program. L_2 is the number of execution levels needed in the extended-Sisal program. L_1/L_2 is a ratio related to the data size of the problem. In this example, $L_1/L_2 = 7/3$. The bigger the data size is, the larger the ratio will be. The performance improvement of the extended-Sisal over the regular Sisal is $(k_1+(k_2+T)*L_1)/(k_1+(k_3)*L_2)$.

	regular-Sisal program		extended-Sisal program	
level	parallelism	nodes visited	parallelism	nodes visited
1	1	1	1	1
2	3	2,3,5	3	2,3,5
3	8	3,4,6,7	8	3,4,6,7
4	15	4,6,7,8,9	The correct result is already found in level 3	
5	15	4,6,7,8,9		
6	7	4,8	The correct result cannot be ensured until the last level: 7 ←	
7	1	4		

Figure 7.15: Statistics of shortest path programs

	regular-Sisal program	extended-Sisal program	
Node visited in all levels	1,2,3,4,5,6,7,8	1,2,3,4,5,6,7	extended-Sisal program does not visit node 8
Number of invocations	1+3+8+15+15+7+1= 50	1+3+8=12	related to resource consumption: storage & processors
Execution time	$k_1+(k_2+T)*L_1$, where $L_1=7$	$k_1+k_3*L_2$, where $L_2=3$	T is the minival searching time~O(logN), N: array size

Figure 7.16: Number of invocations and execution time for shortest path programs

Suppose the data size is large then T >> 0 and L_1 >> L_2. Consequently for big data size the performance improvement of extended-Sisal over regular Sisal program is huge.

7.4 Application: The Puzzle of Colored Blocks

Consider the puzzle which consists of N six-sided cubes, with each side of each cube painted one of N different colors [44]. A solution to the puzzle would consist of an arrangement of the cubes in a row such that on all four sides of the row, all of the N colors is showing. For example, Figure 7.19 is one of many possible solutions for four cubes. The number of possible configuration of N blocks is $(6 * 4)^N = 24^N$ This is a hard problem because of the high number of possible configurations, and the even higher number of paths needed to achieve the goal. It would be unreasonable to solve it by exhaustively trying all possibilities. To overcome the combinatorial explosion, a heuristic search technique called hill climbing is used. Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. We first need to define a *heuristic function* that describes how close a particular configuration is to being a solution. One such function is simply the sum of the number of different colors on each of the four sides. A solution to the puzzle will have a heuristic value of $N * N$. Then we need to define an operation that describes the way of transforming one configuration into another. The operation is simply pick a block (from the N blocks) and rotate it 90 degrees in any directions. For each particular configuration, there are 6*N possible next configurations. Now hill climbing can begin. We generate a new state by selecting a block and rotating it. If the resulting state is better, then keep it. If not, we return to the previous state and try a different perturbation. If implemented in a functional program, all the next steps better than the current state can be perused in parallel *implicitly*. This will allow us to get the results faster

than an imperative program which is executed in sequential mode because of its imperative features.

With extended Sisal 2.0, we can do better than both cases because we can make use of a global variable to indicate the minimum steps the execution model has obtained so far and cut the child threads of a configuration whose heuristic value already exceeds the minimum steps achieved so far. Suppose there is no updatable variable Gmin, then the implementation of the optimization will be very difficult. In the following program, we assume N=4. Figure 7.17 shows the initial configuration of the colored blocks.

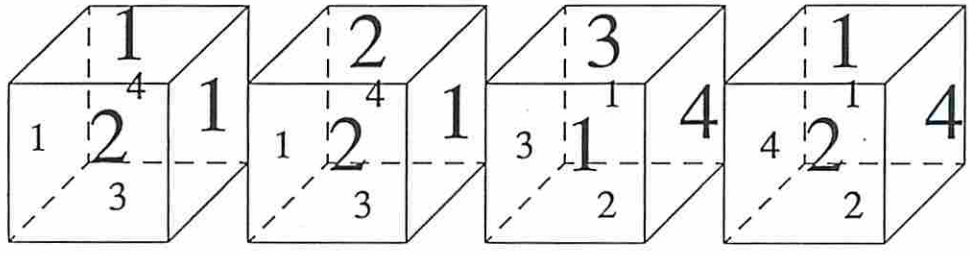

Figure 7.17: Initial configuration of four colored blocks

```
function main(returns PATH)
special Gmin end special; % The only special variable
let
  Gmin := 9999;
  CC := array integer [1..4,1..6:
     [1,..]=1, 2, 3, 4, 1, 1;
     [2,..]=2, 2, 3, 4, 1, 1;
     [3,..]=3, 1, 2, 1, 3, 4;
     [4,..]=1, 2, 2, 1, 4, 4;
                              ];
 in
    LetsRotate(CC,NULL_PATH)
 end let
end function
function LetsRotate(cc: CONFIG, not_yet_goal_path: PATH returns PATH)
let
  current_step_length = calculate_path_length(not_yet_goal_path);
in
% This thread not needed! %
 if current_step_length >= Gmin then NULL_PATH
 else
 for block in [1..4] dot rot in [1..6]
  do
        nc:= rotateOneBlock(block,cc,rot);
```

```
        new_path_added_block_rot:=
             append_block_rot(not_yet_goal_path,block,rot);
  returns
% got goal, the path is a possible result! %
if heuristic_function(nc)=16
then
 let
        one_goal_path_length:=calculate_path_length(
                              new_path_added_block_rot);
% Not allowed in regular Sisal! %
        temp:=  reassign_Gmin(one_goal_path_length)
        in new_path_added_block_rot
  end let
% Not reach goal yet, but is promising... %
elseif heuristic_function(nc) > heuristic_function(cc)
then
        LetsRotate(nc,new_path_added_block_rot)
else
% not goal yet, and not promising %
        returns NULL_PATH
end for
end function
% reassign_Gmin is a special function %
special function reassign_Gmin(V)
          let Gmin := V
             in Gmin
          end let
end function
```

The path found by the program demands 5 rotations: Gmin=5. (b=4,r=1, h=16)(b=4,r=3, h=14)(b=3,r=2, h=12)(b=1,r=2, h=11)(b=1,r=3, h=10) The heuristic value of the initial configuration is 9 since there are 3, 2, 2, and 2 different colors in the four sides of the row of the four block. This path indicates (from right to left): Step one: rotate block 1 in X-to-Z direction and the heuristic value (h) becomes 10 (Figure 7.18.) Step two: rotate block 1 in Y-to-X direction and h becomes 11. Step three: rotate block 3 in Y-to-X direction and h becomes 12. Step four: rotate block 4 in X-to-Z direction and h becomes 14. Step five: rotate block 4 in X-to-Y direction and h becomes 16. The goal configuration is shown in Figure 7.19. Without the optimization using Gmin, 6701 threads are invoked. With the optimization, 3115 threads are invoked. The C program's imperative features hinder the parallel execution. The C compiler must presume a sequential mode. We conclude that this kind of problem is highly suited to be implemented in extended Sisal 2.0. Running

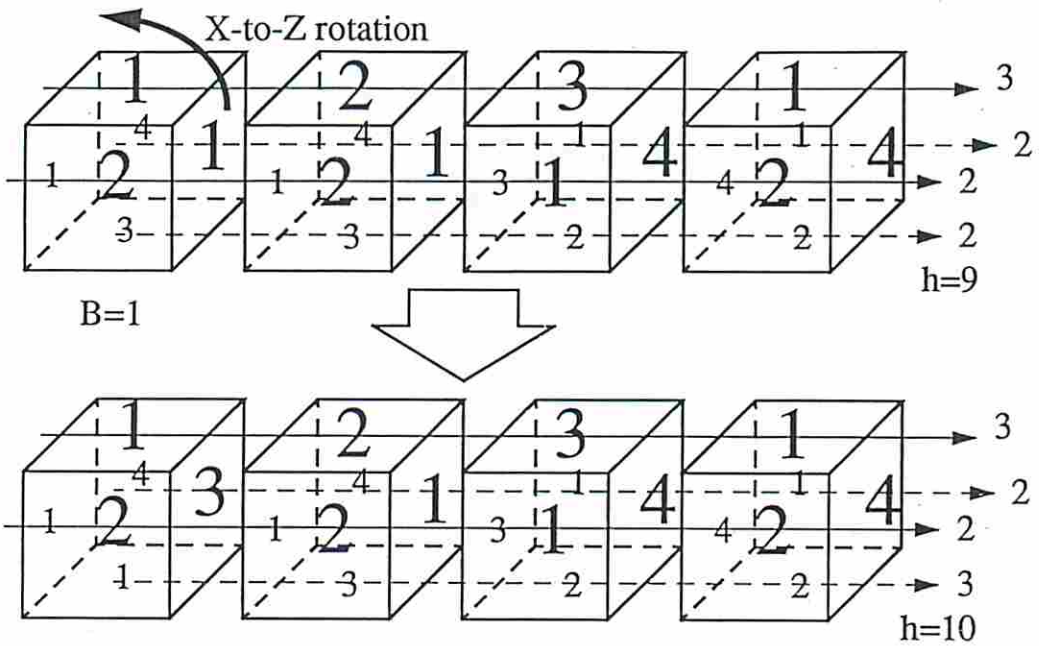

Figure 7.18: One transformation from initial configuration

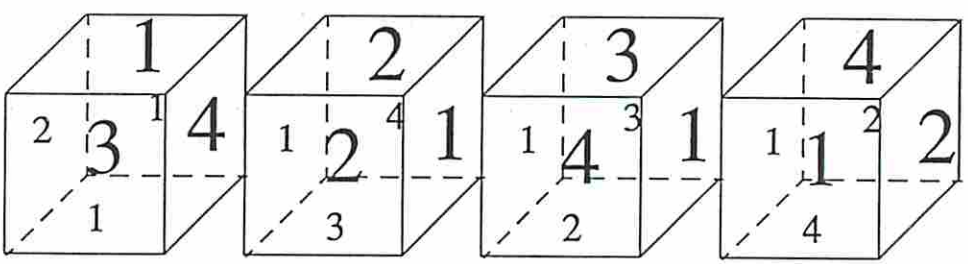

Figure 7.19: Goal configuration of four colored blocks

programs on a Sun SPARCsystem 600 (4/6x0) machine with the assumption of two processors, we got the following performance data: The execution time of the C program is 1.001 msec. The execution time of the regular Sisal program is 1.999 msec. The execution time of the extended Sisal 2.0 program is 0.501 msec. The regular Sisal is the worst because the Gmin optimization cannot be implemented in this language. If more processors are working, regular Sisal can perform better than the C program because of its easy parallel execution.

Chapter 8

Summary and Conclusions

The presence of unnecessary side-effects is regarded as harmful to parallel multi-threaded execution, but some actions leading to side-effects are considered desirable. By extending Sisal with imperative features, our work have developed a language whose semantics fit exactly between a single assignment functional language and an imperative language. It allows stateful computations but also maintains referential transparency and parallel multi-threading. Our work demonstrated that a function calling stateful computations can still be a pure function, and showed how the parallelism in a program can be extracted in a multi-threaded run time system.

The implementation of this work relies heavily on tools in the Centaur system. We have extended the Sisal 2.0 language with user declared special variables. The programmer is allowed to declare certain variables as special. A special variable will be treated as a global variable within the domain of the function inside which the variable has been declared. By default, a variable without being declared as special is an ordinary variable. Programs in extended-Sisal and regular Sisal have been written to demonstrate that with the extension of the special variable scheme it is easier to program and perform parallel computing for various applications in multi-threaded multiprocessor systems. It is found that the language is flexible, easy to use, and able to express the parallelism inherent in the applications.

The main contributions of our work are as follows: (1) Programmers will be able to manipulate stateful computations in a language supporting parallel multi-threaded execution. (2) The imperative features in programs can be easily recognized. (3) Programmers can use various types of special variables to increase the parallelism. (4) The mechanism for the ordinary variable declaration increases the safety and enables the reuse of the variable name. (5) Programmers are still able to program in a purely functional language style when its simplicity and implicit parallelism are desired.

In comparison with purely functional languages, the extended-Sisal has more expressive power and control behaviors due to the availability of stateful computation features. I firmly believe that these advantages will be realized in many applications in addition to the application examples indicated in our work. Since the non-destructive update constraints limit the choice of program structures so far, Lisp and Prolog are the only examples of less imperative languages to have achieved wide use. Although I believe that functional languages will become more widely used in the near future, the value of our work can be justified by observing the fact that most of today's programs really do something useful are those containing some imperative natures.

Appendix A

Typol Programs

The following are the Typol files used in the Centaur for the extended Sisal: environment.ty, for_array_of.ty, sisal_array1.ty, sisal_array11.ty, sisal_array_concat.ty, sisal_array_generate.ty, sisal_array_reference.ty, sisal_array_update.ty, sisal_for_exp.ty, sisal_expression.ty, system_definition.ty.

The use of each file is explained and the contents of files are appended in the following:

A.1 Program Environment

The Typol program *environment.ty* contains the sets of rules which deal with the assignments of both special and ordinary variables.

```
import diff(_,_),  write(_), writeln(_), appendtree(_,_,_) from prolog;
set assignments2 is
judgement  ENV |- DECL_ID_LIST, VALUES : ENV,ENV;
judgement  ENV |- DECL_ID , VALUE -> PAIR,integer;
check_whether_NAME_in_SENV(SENV|- DECL_ID ->false())
& SENV|- DECL_ID, VALUE -> PAIR,1
---------
SENV|- decl_id_list[DECL_ID], values[VALUE] : env[PAIR],SENV;

check_whether_NAME_in_SENV(SENV|- DECL_ID ->true())
& SENV|- DECL_ID, VALUE -> PAIR,2
---------
SENV|- decl_id_list[DECL_ID], values[VALUE] : ENV1, env[PAIR.SENV] ;

SENV|- DECL, VAL -> PAIR, 1 & SENV|- DECLS, VALS : ENV, SENV1
-------------
SENV|- decl_id_list[DECL.DECLS], values[VAL.VALS] : env[PAIR.ENV], SENV1;

SENV|- DECL, VAL -> PAIR, 2 & SENV|- DECLS, VALS : ENV, SENV1
-------------
```

```
SENV|- decl_id_list[DECL.DECLS], values[VAL.VALS] : ENV, env[PAIR.SENV1];

SENV|- function_header(NAME,_,_), CLOSURE -> pair(NAME, CLOSURE), 2;

check_whether_NAME_in_SENV(SENV|-name NAME->true())
-----------------------------------------------
SENV|- name NAME, VAL ->  pair(name NAME, VAL),2;

check_whether_NAME_in_SENV(SENV|-name NAME->false())
-----------------------------------------------
SENV|- name NAME, VAL -> pair(name NAME, VAL),1;
end assignments2 ;
set check_whether_NAME_in_SENV is
judgement ENV |- DECL_ID -> VALUE;
env[] |- DECL_ID -> false();
env[pair(DECL_ID,_)]|- DECL_ID -> true();
env[pair(DECL_ID,_)._] |- DECL_ID -> true();
SENV |- DECL_ID -> VALUE
-------------------------------------
env[pair(DECL_ID',_).SENV] |- DECL_ID -> VALUE; provided diff(DECL_ID', DECL_ID);
end check_whether_NAME_in_SENV;
set assignments is
judgement  |- DECL_ID_LIST, VALUES : ENV;
judgement  |- DECL_ID , VALUE -> PAIR ;
|- DECL_ID, VALUE -> PAIR
---------
|- decl_id_list[DECL_ID], values[VALUE] : env[PAIR] ;

|- DECL, VAL -> PAIR & |- DECLS, VALS : ENV
-------------
|- decl_id_list[DECL.DECLS], values[VAL.VALS] : env[PAIR.ENV] ;

|- function_header(NAME,_,_), CLOSURE -> pair(NAME, CLOSURE) ;

|- name NAME, VAL -> pair(name NAME, VAL);
do write("ASSIGNMENT_NUMBER:") & write(NAME) & writeln(VAL) ;
end assignments ;
set search_in_env0 is
    judgement NAME |- ENV : VALUE,integer;
    NAME |- env[] : FREE_VALUE,0; -- Not found

    NAME |- env[pair(NAME,VALUE).ENV] : VALUE,1;

    name NAME |- ENV : VALUE,INTEGER
    -----------------------------------------------
    name NAME |- env[pair(name NAME1, _).ENV] : VALUE,INTEGER; provided diff(NAME1, NAME);
end search_in_env0;
set search_in_env is
    judgement NAME |- ENV,ENV : VALUE;

    search_in_env0(name NAME |- ENV : VALUE,1)
    -----------------------------------------------
    name NAME |- ENV,SENV : VALUE;

    search_in_env0(name NAME |- SENV : VALUE,1)
    -----------------------------------------------
    name NAME |- ENV,SENV : VALUE;
end search_in_env;
set type_in_envOLD is
judgement NAME |- ENV,ENV : TYPE_SPEC ;
    type(VALUE -> TYPE_SPEC)
    ---------
    NAME |- env[pair(NAME,VALUE).ENV] : TYPE_SPEC;

    name NAME |- ENV : TYPE_SPEC
    -----------------------------------------------
set type_in_envOLD is
judgement NAME |- ENV,ENV : TYPE_SPEC ;
```

```
    type(VALUE -> TYPE_SPEC)
    ---------
    NAME |- env[pair(NAME,VALUE).ENV] : TYPE_SPEC;

    name NAME |- ENV : TYPE_SPEC
    -------------------------------------------
    name NAME |- env[pair(name NAME1, _).ENV] : TYPE_SPEC; provided diff(NAME1, NAME);
end type_in_env ;
set type is
judgement (VALUE -> TYPE_SPEC) ;
(false() -> boolean_type()) ;
(true() -> boolean_type()) ;
(nil() -> null_type()) ;
(int_const _ -> integer_type()) ;
(real_const _ -> real_type()) ;
(double_const _ -> double_type());
(complex_const(_,_) -> complex_type());
(double_complex_const(_,_) -> double_complex_type ());
(character_const _ -> character_type());
(character_string_const _ -> character_type());
(array(_,_) -> array_type(_,_) );
(stream[_.._] -> stream_type(_));
(closure(_,_,_,_) -> function_type(_,_)) ;
end type ;
end environment;
```

A.2 Program For_array_of

The Typol program *for_array_of.ty* consists the set of rules which handle the "array of" semantics.

```
use sisal;
use PSP;
import appendtree(_,_,_),diff(_,_) from prolog;
import
       plus(_,_,_), plus1(_,_), uminus(_,_), lt(_,_,_),star(_,_,_)
from functions;
import type(VALUE -> TYPE_SPEC)
from environment;
import
 build(SYSTEM,ENV,EXPRESSION|-SIZE_DESCR_LIST->VALUE,integer)
from sisal_array_generate;
import eval_expression(SYSTEM, ENV,ENV |- EXPRESSION :  VALUES,ENV)
from sisal_expression;
set handle_array_of is
judgement SYSTEM,ENV,VALUE |-SIZE_DESCR_LIST,EXPRESSION,FILTER_OR_AT_LIST:VALUE;

SYSTEM,ENV,OLD_VAL |- size_descr_list[],_,no_expression(): OLD_VAL;

remake_array_with_size_descr_list(ENV,OLD_ARRAY|-SIZE_DESCR_LIST->NEW_ARRAY)
-------------------------------------------------------------------------------
SYSTEM,ENV,OLD_ARRAY|-SIZE_DESCR_LIST,_,no_expression():NEW_ARRAY;
end handle_array_of;
set remake_array_with_size_descr_list is
 judgement ENV,VALUE |- SIZE_DESCR_LIST -> VALUE;

fill_one_dim_array(OLD_ARRAY|-TRIPLET2->NEW_ARRAY)
--------------------------------------------------
ENV,OLD_ARRAY |- size_descr_list[TRIPLET2]->NEW_ARRAY;

build(_,_,_ |- size_descr_list[TRIPLET2.SDL]->DEF_ARRAY,DIM)
```

```
& fill_array2(ENV,OLD_ARRAY |- DEF_ARRAY -> NEW_ARRAY)
-------------------------------------------------------------
ENV,OLD_ARRAY |- size_descr_list[TRIPLET2.SDL]->NEW_ARRAY;
end remake_array_with_size_descr_list;
set fill_one_dim_array is
 judgement VALUE|-TRIPLET2->VALUE;

 elt_to_elts2(OLD_ARRAY->D,_,_,ELTS)
& elts_to_elt2(D,I1,upper I2,ELTS->NEW_ARRAY)
-----------------------------------------------------------------
 OLD_ARRAY |- triplet2(_, triplet(int_const I1,int_const I2, _))->NEW_ARRAY;
end fill_one_dim_array;
set fill_array2 is
judgement ENV,VALUE |- VALUE -> VALUE;

 elt_to_elts2(OLD_ARRAY->_,_,_,OLD_ELTS)
& elt_to_elts2(DEF_ARRAY->D,L,U,DEF_ELTS) &
 fill_ELTS_into_def_ELTS(ENV,OLD_ELTS |- DEF_ELTS ->NEW_ELTS, REM_ELTS)
 & elts_to_elt2(D,L,U, NEW_ELTS->NEW_ARRAY)
-----------------------------------------------------------------
 ENV,OLD_ARRAY |- DEF_ARRAY ->NEW_ARRAY;
end fill_array2;
set fill_ELTS_into_def_ELTS is
judgement ENV,ELTS |- ELTS -> ELTS, ELTS;

fill_one_arr(ENV,ELTS |- DEF_ELT : NEW_ELT, REM_ELTS)
-----------------------------------------------------
ENV,ELTS |-  elts[DEF_ELT] -> elts[NEW_ELT], REM_ELTS;

fill_one_arr(ENV,ELTS |-  DEF_ELT : NEW_ELT, REM_ELTS)
& ENV,REM_ELTS |-  DEF_ELTS -> NEW_ELTS, REM_ELTS'
-------------------------------------------------
ENV,ELTS |-  elts[DEF_ELT.DEF_ELTS] -> elts[NEW_ELT.NEW_ELTS], REM_ELTS';
end fill_ELTS_into_def_ELTS;
set fill_one_arr is
judgement ENV,ELTS |- ELT : ELT, ELTS;
elt_to_elts2(array(DIM,DIM_CONTENT)->D,L,U,ELTS2)
& fill_elts2(ENV,ELTS |- ELTS2 -> ELTS2', REM_ELTS) &
elts_to_elt2(D,L,U,ELTS2'-> NEW_ELT)
-------------------------------------------------
ENV,ELTS |- array(DIM,DIM_CONTENT): NEW_ELT, REM_ELTS;
end fill_one_arr;
set fill_elts2 is
 judgement ENV,ELTS |- ELTS -> ELTS, ELTS;
type(THIS_ELT -> array_type(_,_))
& fill_one_arr(ENV,ELTS |- THIS_ELT : NEW_THIS_ELT, REM_ELTS)
-------------------------------------------------------------
ENV,ELTS |- elts[THIS_ELT] -> elts[NEW_THIS_ELT], REM_ELTS;

type(THIS_ELT -> array_type(_,_))
& fill_one_arr(ENV,ELTS |- THIS_ELT : NEW_THIS_ELT, REM_ELTS) &
ENV,REM_ELTS |- ELTS2 -> ELTS3, REM_ELTS'
-------------------------------------------------------------
ENV,ELTS |- elts[THIS_ELT.ELTS2] -> elts[NEW_THIS_ELT.ELTS3], REM_ELTS';

ENV,ELTS |- elts[] -> elts[], ELTS;

ENV,elts[ELT.ELTS] |- elts[THIS_ELT] -> elts[ELT], ELTS;

ENV,ELTS |- ELTS2 -> ELTS3, REM_ELTS
-------------------------------------------------------------
ENV,elts[ELT.ELTS] |- elts[THIS_ELT.ELTS2] -> elts[ELT.ELTS3], REM_ELTS;
end fill_elts2;
set handle_reduction_or_array_of is
 judgement  SYSTEM, ENV, VALUES |- RETURN_LIST -> VALUES;
 judgement  SYSTEM, ENV, VALUE |- RETURN : VALUE;
    SYSTEM, ENV, OLD_VAL |- RETURN : NEW_VAL
    ----------------------------------------
```

```
    SYSTEM, ENV, values[OLD_VAL] |- return_list[RETURN] -> values[NEW_VAL];

    SYSTEM, ENV, OLD_VAL |- RETURN : NEW_VAL
& SYSTEM, ENV, VALS |- RETURNS -> NEW_VALS
    ------------------------------------------
    SYSTEM, ENV, values[OLD_VAL.VALS]
|- return_list[RETURN.RETURNS] -> values[NEW_VAL.NEW_VALS];

    plus_list(|-STREAM->SUM)
    ------------------------------------------------------------------------
    SYSTEM, ENV, STREAM |- return_value(invocation(name "sum",PARAMS),_) : SUM;

    prod_list(|- LIST_VAL -> PRODUIT)
    ---------------------------------
    SYSTEM, ENV, LIST_VAL |- return_value(invocation(name "product",PARAMS),_) : PRODUIT;

    min_list(|- LIST_VAL -> MINIMUM)
    ----------------
    SYSTEM, ENV, LIST_VAL |-return_value(invocation(name "minval", PARAMS),_) : MINIMUM;

    max_list(|- LIST_VAL -> MAXIMUM)
    ----------------
    SYSTEM, ENV, LIST_VAL |-return_value(invocation(name "maxval", PARAMS),_) : MAXIMUM;

    SYSTEM, ENV, OLD_VAL |- return_value(Exp,F) : OLD_VAL;

end handle_reduction_or_array_of;
set eval_return2 is
    judgement SYSTEM, ENV,ENV, VALUES |- RETURN_LIST ->VALUES,ENV;
    judgement SYSTEM, ENV,ENV, VALUE |- RETURN : VALUE,ENV;
elt_to_elts2(OLD_ARRAY->D,L,upper OLD_U,OLD_ELTS)
 & eval_expression(SYSTEM,ENV,SENV|-EXP:values[VAL],NEW_SENV) &
pass_when_criteria(SYSTEM,ENV,NEW_SENV|-BINARY_REL_EXPRESSION->true()) &
appendtree(OLD_ELTS, elts[VAL], NEW_ELTS) & plus1(OLD_U,NEW_U)
 & elts_to_elt2(dim 1 ,L,upper NEW_U,NEW_ELTS->NEW_ARRAY)
-------------------------------------------------------------------------
SYSTEM,ENV,SENV,OLD_ARRAY
|- return_array(_,EXP,when_expression(BINARY_REL_EXPRESSION))
 :NEW_ARRAY,NEW_SENV;

elt_to_elts2(OLD_ARRAY->D,L,upper OLD_U,OLD_ELTS)
& eval_expression(SYSTEM,ENV,SENV|-EXP:values[VAL],NEW_SENV) &
pass_when_criteria(SYSTEM,ENV,NEW_SENV|-BINARY_REL_EXPRESSION->false())
-------------------------------------------------------------------------
SYSTEM,ENV,SENV,OLD_ARRAY |-
return_array(_,EXP,when_expression(BINARY_REL_EXPRESSION)) :OLD_ARRAY,NEW_SENV;

elt_to_elts2(OLD_ARRAY->D,L,upper OLD_U,OLD_ELTS)
& eval_expression(SYSTEM,ENV,SENV|-EXP:values[VAL],NEW_SENV) &
appendtree(OLD_ELTS, elts[VAL], NEW_ELTS) &
plus1(OLD_U,NEW_U) & elts_to_elt2(dim 1 ,L,upper NEW_U,NEW_ELTS->NEW_ARRAY).
-------------------------------------------------------------------------
SYSTEM,ENV,SENV,OLD_ARRAY|- return_array(_,EXP,_):NEW_ARRAY,NEW_SENV;

  SYSTEM, ENV, SENV,OLD_ELT |- RETURN : NEW_ELT,NEW_SENV
  ------------------------------------------------------------------------
  SYSTEM, ENV, SENV,values[OLD_ELT]
|- return_list[RETURN] -> values[NEW_ELT],NEW_SENV;

  SYSTEM, ENV, SENV, OLD_ELT |- RETURN : NEW_ELT,NEW_SENV' &
  SYSTEM, ENV, NEW_SENV',OLD_ELTS |- RETURNS -> NEW_ELTS,NEW_SENV
  ------------------------------------------------------------
  SYSTEM, ENV, SENV,values[OLD_ELT.OLD_ELTS]
 |- return_list[RETURN.RETURNS] -> values[NEW_ELT.NEW_ELTS],NEW_SENV;

eval_expression(SYSTEM,ENV,SENV|-EXP:values[VAL],NEW_SENV)
& appendtree(OLD_STREAM, stream[VAL], NEW_STREAM)
-------------------------------------------------
```

```
SYSTEM,ENV,SENV,OLD_STREAM |-
 return_value(invocation(name "sum",expression_list[EXP]),_):
NEW_STREAM,NEW_SENV;

eval_expression(SYSTEM,ENV,SENV|-EXP:values[VAL],NEW_SENV)
& appendtree(OLD_STREAM, stream[VAL], NEW_STREAM)
-----------------------------------------------------
SYSTEM,ENV,SENV,OLD_STREAM |-
 return_value(invocation(name "product",expression_list[EXP]),_):
NEW_STREAM,NEW_SENV;

eval_expression(SYSTEM,ENV,SENV|-EXP:values[VAL],NEW_SENV)
& appendtree(OLD_STREAM, stream[VAL], NEW_STREAM)
-----------------------------------------------------
SYSTEM,ENV,SENV,OLD_STREAM
|- return_value(invocation(name "maxval",expression_list[EXP]),_) :
NEW_STREAM,NEW_SENV;

eval_expression(SYSTEM,ENV,SENV|-EXP:values[VAL],NEW_SENV)
& appendtree(OLD_STREAM, stream[VAL], NEW_STREAM)
-----------------------------------------------------
SYSTEM,ENV,SENV,OLD_STREAM
|- return_value(invocation(name "minval",expression_list[EXP]),_)
 : NEW_STREAM,NEW_SENV;

 eval_expression(SYSTEM, ENV,SENV |- EXP : values[NEW_VAL],NEW_SENV)
 ------------------------------------------------------------
 SYSTEM,ENV,SENV,OLD_ELT|- return_value(EXP,F) : NEW_VAL,NEW_SENV;
end eval_return2;
set pass_when_criteria is
 judgement SYSTEM,ENV,ENV|-EXPRESSION->VALUE;
 eval_expression(SYSTEM,ENV,SENV|-BINARY_REL_EXPRESSION:values[VALUE],_)
-------------------------------------------------------
 SYSTEM,ENV,SENV|-BINARY_REL_EXPRESSION -> VALUE;
end pass_when_criteria;
set elt_to_elts2 is
 judgement (ELT -> DIM,integer,UPPER,ELTS);
 (array(D,dim_content(lower LC,U,ELTS)) -> D,LC,U,ELTS);
end elt_to_elts2;
set elts_to_elt2 is
 judgement (DIM,integer,UPPER,ELTS ->ELT);
           (D,LC,U, ELTS-> array(D,dim_content(lower LC,U,ELTS)));
end elts_to_elt2;
end sisal_for_exp;
```

A.3 Program Sisal_array1

The Typol program *sisal_array1.ty* is a collection of rule sets which deal manipulate the array operations.

```
use sisal;
import diff(_,_) from prolog;
import eval_expression_list(SYSTEM, ENV,ENV |- EXPRESSION_LIST ->  VALUES, ENV),
     eval_expression(SYSTEM, ENV,ENV |- EXPRESSION :  VALUES,ENV)
from sisal_expression;
import plusSTEP(integer,OPT_EXPRESSION:integer)
from sisal_array_reference;
import search_in_env(NAME |- ENV,ENV : VALUE) from environment;
import
  extractP1Vals(SYSTEM,ENV|-ARRAY_PART->integer,VALUES)
from sisal_array_generate;
```

```
set modifyPlarray is
 judgement |- VALUES,VALUES,VALUE -> VALUE;
 elt_to_elts(ARRAY->D,L,U,ELTS)
&modifyPelts(|-PL,L,VALUES,ELTS->ELTS') &elts_to_elt(D,L,U,ELTS'->ARRAY')
-------------------------------------------------------------------------
  |-PL,VALUES,ARRAY->ARRAY';
end modifyPlarray;
set modifyPelts is
 judgement |- VALUES,integer,VALUES,ELTS -> ELTS;
 |- values[int_const P],P,values[ELT'],elts[ELT]->elts[ELT'];

 |- values[int_const P],P,values[ELT'],elts[ELT.ELTS]->elts[ELT'.ELTS];

   dim_content_or_elt_to_elts(ELT->D,L,U,ES') & |- PL,L,VALUES,ES'->ES''
& elts_to_elt(D,L,U,ES''->ELT')
-------------------------------------------------------------
 |- values[int_const P.PL],P,VALUES,elts[ELT]->elts[ELT'];

   dim_content_or_elt_to_elts(ELT->D,L,U,ES') & |- PL,L,VALUES,ES'-> ES''
& elts_to_elt(D,L,U,ES''->ELT')
-----------------------------------------------------------------
 |- values[int_const P.PL],P,VALUES,elts[ELT.ELTS]->elts[ELT'.ELTS];

  plus1(P',P''') & |- values[int_const P.PL],P''',VALUES1,ELTS->ELTS'
-------------------------------------------------------------------
 |-values[int_const P.PL],P',VALUES1,elts[ELT1.ELTS]->elts[ELT1.ELTS'];
 provided diff(P,P');
end modifyPelts;
set dim_content_or_elt_to_elts is
 judgement ( ELT -> DIM,integer,UPPER,ELTS);
 (array(D,dim_content(lower LC,U,ELTS)) -> D,LC,U,ELTS);
 (dim_content(lower LC,U,ELTS) -> dim 1,LC,U,ELTS);
end dim_content_or_elt_to_elts;
set elt_to_elts is
 judgement ( ELT -> DIM,integer,UPPER,ELTS);
 (array(D,dim_content(lower LC,U,ELTS)) -> D,LC,U,ELTS);
end elt_to_elts;
set elts_to_elt is
 judgement (DIM,integer,UPPER,ELTS ->ELT);
            (D,LC,U, ELTS-> array(D,dim_content(lower LC,U,ELTS)));
end elts_to_elt;
set modify_place_of_elts is
 judgement |- integer,integer,VALUES,ELTS -> ELTS;
 |- PLACEMENT,PLACEMENT,values[ELT'],elts[ELT] -> elts[ELT'];

 |- PLACEMENT,PLACEMENT,values[ELT'],elts[ELT.ELTS] -> elts[ELT'.ELTS];

  diff(PLACEMENT,INDEX) & plus1(INDEX,INDEX')
& |- PLACEMENT,INDEX', VALUES1, ELTS -> ELTS'
-----------------------------------------------------------------------
  |- PLACEMENT,INDEX,VALUES1,elts[ELT1.ELTS] -> elts[ELT1.ELTS'];
end modify_place_of_elts;
set extract_pattern_values is
    judgement SYSTEM, ENV,ENV |-PLACEMENT,EXPRESSION_LIST
-> VALUES, integer,VALUES,ELTS,ENV;

    eval_expression_list(SYSTEM, ENV,SENV |- EXP_LIST ->  VALUES,SENV')
    & parse_SPL(SYSTEM,ENV,_|- SPL-> PLIST,PATTERN,ELTS)
    ----------------------------------------------------------------
    SYSTEM,ENV,SENV |- selector(SPL,DSL),EXP_LIST
-> PLIST,PATTERN,VALUES,ELTS,SENV';
end extract_pattern_values;
set parse_SPL is
judgement SYSTEM,ENV,ENV|- SELECTOR_PART_LIST-> VALUES,integer,ELTS;
  SYSTEM,ENV,SENV|-selector_part_list[int_const P] -> values[int_const P],1,elts[];

     search_in_env(ArrayName|-ENV,SENV:IndexArray)
& getIndexELTS(|-IndexArray->IndexELTS)
```

```
    ------------------------------------------------------------
    SYSTEM,ENV,SENV|-selector_part_list[ArrayName]-> values[],3,IndexELTS;

    eval_expression(SYSTEM,ENV,SENV|-LL_EXP:values[LL],_)
& eval_expression(SYSTEM,ENV,SENV|-UU_EXP:values[UU],_)&
    LLUUSTEPtoIndexELTS(LL,UU,STEP->IndexELTS)
 ------------------------------------------------------------------
    SYSTEM,ENV,SENV|-
selector_part_list[triplet2(NN, triplet(LL_EXP,UU_EXP,STEP))]
-> values[],4,IndexELTS;

   SYSTEM,ENV,SENV|-SPL->VALS,PATTERN,ELTS
 ----------------------------------------------------------
   SYSTEM,ENV,SENV|-selector_part_list[int_const P.SPL]
-> values[int_const P.VALS],PATTERN,ELTS;

   SYSTEM,ENV,SENV|-SPL->VALS,PATTERN,ELTS
 ----------------------------------------------------------
   SYSTEM,ENV,SENV|-selector_part_list[int_const P.SPL]
-> values[int_const P.VALS],PATTERN,ELTS;
end parse_SPL;
set LLUUSTEPtoIndexELTS is
 judgement (VALUE,VALUE,OPT_EXPRESSION->ELTS);

 (LL,LL,STEP->elts[LL]);

 plusSTEP2(LL,STEP:LL')& (LL',UU,STEP->ELTS)
------------------------------------------------------------
 (LL,UU,STEP->elts[LL.ELTS]); provided diff(LL,UU);
end LLUUSTEPtoIndexELTS;
set plusSTEP2 is
judgement (VALUE,OPT_EXPRESSION:VALUE);
plus1(L,L')
-----------------------------------
(int_const L,no_expression():int_const L');

plus1(L,L')
-----------------------------------
(int_const L,no_expression():int_const L');

plusSTEP(L,STEP:L')
-----------------------------------
(int_const L,STEP:int_const L');

plusSTEP(L,STEP:L')
-----------------------------------
(int_const L,STEP:int_const L');
end plusSTEP2;
set getIndexELTS is
 judgement |- VALUE -> ELTS; |- array(D, dim_content(L,U,IndexELTS))->IndexELTS;
end getIndexELTS;
set modifyOneDimArrByPlaceVals is
 judgement |-integer,VALUES,VALUE->VALUE;
 modify_place_of_elts(|-Place,LOWER,VALUES1,ELTS->ELTS')
-------------------------------------------------------------------------
 |-Place,VALUES1,array(DIM,dim_content(lower LOWER,UPPER,ELTS))
 ->array(DIM,dim_content(lower LOWER,UPPER,ELTS'));
end modifyOneDimArrByPlaceVals;
set modifyOneDimArrByAPL is
judgement SYSTEM,ENV|-ARRAY_PART_LIST,VALUE->VALUE;

  extractP1Vals(SYSTEM,ENV|-ARRAY_PART1->Place,Vals)
& modifyOneDimArrByPlaceVals(|-Place,Vals,ARRAY''->ARRAY)
-------------------------------------------------------------------------
SYSTEM,ENV |- array_part_list[ARRAY_PART1],ARRAY''-> ARRAY;

 extractP1Vals(SYSTEM,ENV|-ARRAY_PART1->Place,VALUES1)
& modifyOneDimArrByPlaceVals(|-Place,VALUES1,ARRAY''->ARRAY')
```

```
& SYSTEM,ENV|-ARRAY_PARTS,ARRAY'->ARRAY
-------------------------------------------------------------------------
 SYSTEM,ENV|-array_part_list[ARRAY_PART1.ARRAY_PARTS],ARRAY''-> ARRAY;
end modifyOneDimArrByAPL;
set liml_execution is
judgement SYSTEM, ENV, ENV |- EXPRESSION_LIST : VALUES;

search_in_env(ArrName |- ENV,SENV: Array) & elt_to_elts(Array->D,L,U,ELTS)
-----------------------------------------------------
SYSTEM, ENV, SENV |- expression_list[ArrName]: values[int_const L];

search_in_env(ArrName |- ENV,SENV: Array)
& go_to_find_liml(SYSTEM, ENV, Array |- DIM -> L)
-------------------------------------------------
SYSTEM, ENV, SENV |- expression_list[ArrName,int_const DIM]
 : values[int_const L];
end liml_execution;
set go_to_find_liml is
judgement SYSTEM, ENV, VALUE |- integer -> integer;

elt_to_elts(Array->D,L,U,ELTS)
-----------------------------------------------
SYSTEM, ENV, Array |- 1 -> L;

diff(DIM,1) & elt_to_elts(Array->D,L1,U,elts[ELT._])
& minus(DIM,1,DIM') & SYSTEM,ENV,ELT |- DIM' -> L2
-------------------------------------------------
SYSTEM, ENV,Array|- DIM -> L2;
end go_to_find_liml;
set limh_execution is
judgement SYSTEM, ENV,ENV |- EXPRESSION_LIST : VALUES;

search_in_env(ArrName |- ENV,SENV: Array) & elt_to_elts(Array->_,_,upper U,_)
-----------------------------------------------------
SYSTEM, ENV,SENV |- expression_list[ArrName]: values[int_const U];

search_in_env(ArrName |- ENV,SENV: Array)
& go_to_find_limh(SYSTEM, ENV, Array |- DIM -> U)
-------------------------------------------------
SYSTEM, ENV,SENV |- expression_list[ArrName,int_const DIM]
: values[int_const U];
end limh_execution;
set go_to_find_limh is
judgement SYSTEM, ENV, VALUE |- integer -> integer;

elt_to_elts(Array->_,_,upper U,_)
-----------------------------------------------
SYSTEM, ENV, Array |- 1 -> U;

diff(DIM,1) & elt_to_elts(Array->_,_,_,elts[ELT._])
& minus(DIM,1,DIM') & SYSTEM,ENV,ELT |- DIM' -> U
-------------------------------------------------
SYSTEM, ENV,Array|- DIM -> U;
end go_to_find_limh;
set size_execution is
judgement SYSTEM, ENV, ENV |- EXPRESSION_LIST : VALUES;

search_in_env(ArrName |- ENV,SENV: Array) & size_of_whole_array(Array->SIZE)
-------------------------------------------------------------------
SYSTEM, ENV, SENV |- expression_list[ArrName]: values[int_const SIZE];

search_in_env(ArrName |- ENV,SENV: Array)
& size_for_this_dim(SYSTEM, ENV, Array |- DIM -> SIZE)
-------------------------------------------------
SYSTEM, ENV, SENV |- expression_list[ArrName,int_const DIM]
: values[int_const SIZE];
end size_execution;
set size_for_this_dim is
```

```
judgement SYSTEM, ENV, VALUE |- integer -> integer;

  elt_to_elts(Array->_,L,upper U,_) & cal_size(L,U,SIZE)
-------------------------------------------------
SYSTEM, ENV, Array |- 1 -> SIZE ;

diff(DIM,1) & elt_to_elts(Array->_,_,_,elts[ELT._])
& minus(DIM,1,DIM') & SYSTEM,ENV,ELT |- DIM' -> SIZE
-------------------------------------------------
SYSTEM, ENV,Array|- DIM -> SIZE;
end size_for_this_dim;
set size_of_whole_array is
judgement (VALUE  -> integer);

(array(dim 0,_)-> 0);

elt_to_elts(Array->dim 1,L,upper U,_) & cal_size(L,U,SIZE1)
-------------------------------------------------
(Array -> SIZE1);

elt_to_elts(Array->dim DIM,L,upper U,elts[ELT._])
& diff(DIM,1) & cal_size(L,U,SIZE2) & (ELT -> SIZE1)
& time(SIZE1,SIZE2,SIZE)
-----------------------------------------
(Array-> SIZE);
end size_of_whole_array;
end sisal_array1;
```

A.4 Program Sisal_array11

The Typol program *sisal_array*11.*ty* contains the sets of rules which manipulate the array related operations such as updates, selection, etc.

```
import constructOneDimEmpArr(EXPRESSION |- integer, integer -> VALUE)
from sisal_array_generate;
import eval_expression_list(SYSTEM, ENV,ENV |- EXPRESSION_LIST ->  VALUES,ENV),
      eval_expression(SYSTEM, ENV,ENV |- EXPRESSION :  VALUES,ENV)
from sisal_expression;
set fill_array is
judgement SYSTEM,ENV,ENV,integer,VALUE|- ARRAY_PART_LIST->VALUE;
 modify_array_with_array_part(SYSTEM,ENV,SENV,ARRAY_PART_COUNT,ARRAY
|-ARRAY_PART->ARRAY')
-----------------------------------------------------------------------
SYSTEM,ENV,SENV,ARRAY_PART_COUNT,ARRAY|- array_part_list[ARRAY_PART]-> ARRAY';
end fill_array;
set modify_array_with_array_part is
judgement SYSTEM,ENV,ENV,integer,VALUE|- ARRAY_PART->VALUE;
 modify_array_with_opt_placement(SYSTEM,ENV,SENV,APC
|- OPT_PLACEMENT,EXPRESSION_LIST,ARRAY->ARRAY')
 ------------------------------------------------------------
 SYSTEM,ENV,SENV,APC,ARRAY|- array_part(OPT_PLACEMENT,EXPRESSION_LIST) ->ARRAY';
end modify_array_with_array_part;
set modify_array_with_opt_placement is
judgement SYSTEM,ENV,ENV,integer|- OPT_PLACEMENT,EXPRESSION_LIST,VALUE->VALUE;
modify_array_with_no_placement(SYSTEM,ENV,SENV,APC
 |- EXPRESSION_LIST,ARRAY->ARRAY')
-----------------------------------------------------------------------
SYSTEM,ENV,SENV,APC|- no_placement(),EXPRESSION_LIST,ARRAY->ARRAY';

modify_array_with_placement(SYSTEM,ENV,SENV
 |-PLACEMENT,EXPRESSION_LIST,ARRAY->ARRAY')
```

```
-----------------------------------------------------------------
SYSTEM,ENV,SENV,APC|-
 PLACEMENT,EXPRESSION_LIST,ARRAY->ARRAY';
provided diff(PLACEMENT,no_placement());
end modify_array_with_opt_placement;
set modify_array_with_placement is
judgement SYSTEM,ENV,ENV|- PLACEMENT,EXPRESSION_LIST,VALUE->VALUE;
 SYSTEM,ENV,SENV|- otherwise(),EXPS,ARRAY-> ARRAY;

    eval_expression_list(SYSTEM,ENV,_|-EXPS->VALUES,_)
& eval_expression(SYSTEM,ENV,_|-L1_EXP:values[L1],_)&
    eval_expression(SYSTEM,ENV,_|-U1_EXP:values[U1],_)
& eval_expression(SYSTEM,ENV,_|-L2_EXP:values[L2],_)&
    eval_expression(SYSTEM,ENV,_|-U2_EXP:values[U2],_)
& LLUUSTEPtoIndexELTS(L1,U1,STEP->IndexELTS1) &
    LLUUSTEPtoIndexELTS(L2,U2,STEP->IndexELTS2)
& modify_array_by_IndexELTS12_by_one_value(IndexELTS1,IndexELTS2
 |- values[],VALUES,ARRAY->ARRAY')
-----------------------------------------------------------------
    SYSTEM,ENV,SENV|-selector(selector_part_list[triplet2(I_EXP, triplet(L1_EXP,U1_EXP,STEP)),
     triplet2(J_EXP,triplet(L2_EXP,U2_EXP,STEP))],_),EXPS,ARRAY-> ARRAY';

end modify_array_with_placement;

set modify_array_with_no_placement is
judgement SYSTEM,ENV,ENV,integer|- EXPRESSION_LIST,VALUE->VALUE;

  elt_to_elts(|- ARRAY->D,L,U,ELTS)
& modify_elts_by_exp_list(SYSTEM,ENV,APC,L|-EXPRESSION_LIST,ELTS-> ELTS')
& elts_to_elt(D,L,U,ELTS'->ARRAY')
-----------------------------------------------------
 SYSTEM,ENV,SENV,APC|- EXPRESSION_LIST,ARRAY->ARRAY';
end modify_array_with_no_placement;

set modify_elts_by_exp_list is
judgement SYSTEM, ENV, integer, integer |- EXPRESSION_LIST,ELTS-> ELTS;

   eval_expression_list(SYSTEM, ENV,_ |- EXP_LIST ->  VALUES,_)
& plus(LOWER,APC,P)
 & modify_elts_by_p1_ByElts(|- P,LOWER,VALUES,ELTS->ELTS')
-----------------------------------------------------------------------
 SYSTEM, ENV, APC, LOWER |- EXP_LIST, ELTS-> ELTS';
end modify_elts_by_exp_list;
set modify_elts_by_p1_ByElts is
judgement |- integer, integer, VALUES,ELTS->ELTS;
  modifyP1arrayByElts(|-VALS,ARR1->ARR1')
-----------------------------------------------------------------------
 |- P1,P1,VALS,elts[ARR1]->elts[ARR1'];

  modifyP1arrayByElts(|-VALS,ARR1->ARR1')
-----------------------------------------------------------------------
 |- P1,P1,VALS,elts[ARR1.ELTS]->elts[ARR1'.ELTS];

  diff(P1,P1') & plus1(P1',P1'') & |-P1,P1'',VALS,ELTS -> ELTS'
-----------------------------------------------------------------------
 |- P1,P1',VALS,elts[ELT1.ELTS]->elts[ELT1.ELTS'];
end modify_elts_by_p1_ByElts;
set update is
judgement ELTS |- VALUES,integer,VALUES,VALUE->VALUE;
  modify_array_by_p12(|- PLIST,VALUES,ARRAY->ARRAY')
-----------------------------------------------------------------------
 elts[] |-PLIST,1,VALUES,ARRAY-> ARRAY';

  modifyArrByArr(|-PLIST,VALUES1,ARRAY'-> ARRAY)
-----------------------------------------------------------------------
 elts[] |-PLIST,2,VALUES1,ARRAY'-> ARRAY;

 modify_array_by_IndexELTS(IndexELTS|- PLIST,VALUES1,ARRAY->ARRAY')
```

```
------------------------------------------------------------------------
 IndexELTS |- PLIST,3,VALUES1,ARRAY-> ARRAY';

modify_array_by_IndexELTS_by_one_value(IndexELTS|- PLIST,VALUES1,ARRAY->ARRAY')
------------------------------------------------------------------------
 IndexELTS |- PLIST,4,VALUES1,ARRAY-> ARRAY';
end update;
set modify_array_by_IndexELTS is
judgement ELTS |- VALUES, VALUES, VALUE -> VALUE;

   append_at_end(PL,P2->PL2)
& modify_array_by_p12(|- PL2,values[VAL],ARRAY->ARRAY')
------------------------------------------------------------------------
 elts[int_const P2] |- PL,values[VAL],ARRAY->ARRAY';

   append_at_end(|- PL,P2->PL2)
& modify_array_by_p12(|- PL2,values[VAL],ARRAY->ARRAY')
& IndexELTS|-PL,VALUES,ARRAY'->ARRAY''
------------------------------------------------------------------------
 elts[int_const P2.IndexELTS] |- PL,values[VAL.VALUES],ARRAY->ARRAY'';
end modify_array_by_IndexELTS;
set modify_array_by_IndexELTS_by_one_value is
judgement ELTS |- VALUES, VALUES, VALUE -> VALUE;

   append_at_end(PL,P2->PL2) &
 modify_array_by_p12(|- PL2,values[VAL],ARRAY->ARRAY')
------------------------------------------------------------------------
 elts[int_const P2] !- PL,values[VAL],ARRAY->ARRAY';

end modify_array_by_IndexELTS_by_one_value;
set modify_array_by_IndexELTS12_by_one_value is
judgement ELTS,ELTS |- VALUES,VALUES, VALUE -> VALUE;

   append_at_end(PL,P1->PL2) & append_at_end(PL2,P2->PL2')
 & modify_array_by_p12(|- PL2',values[VAL],ARRAY->ARRAY')
------------------------------------------------------------------------
 elts[int_const P1],elts[int_const P2] |- PL,values[VAL],ARRAY->ARRAY';

end modify_array_by_IndexELTS12_by_one_value;
set append_at_end is
judgement (VALUES,integer->VALUES);

(values[],P2->values[int_const P2]);

(PL,P2->PL2)
-------------------------------
(values[P.PL],P2->values[P.PL2]);
end append_at_end;
set modify_elts_by_p1_array is
judgement |- VALUES, integer, VALUES,ELTS->ELTS;

  modifyP1arrayByArray(|-VALS,ARR1->ARR1')
------------------------------------------------------------------------
 |- values[int_const P],P,VALS,elts[ARR1]->elts[ARR1'];

  modifyP1arrayByArray(|-VALS,ARR1->ARR1')
------------------------------------------------------------------------
 |- values[int_const P],P,VALS,elts[ARR1.ELTS]->elts[ARR1'.ELTS];

 elt_to_elts(ELT1->D,L,U,ES) & |-PL,L,VALS,ES -> ES'
& elts_to_elt(D,L,U,ES'->ELT1')
------------------------------------------------------------------------
 |- values[int_const P.PL],P,VALS,elts[ELT1.ELTS]->elts[ELT1'.ELTS];

   plus1(P',P'') & |-values[int_const P.PL],P'',VALS,ELTS -> ELTS'
------------------------------------------------------------------------
 |- values[int_const P.PL],P',VALS,elts[ELT1.ELTS]->elts[ELT1.ELTS'];
provided diff(P,P');
```

```
end modify_elts_by_p1_array;
set modifyP1arrayByElts is
judgement |- VALUES,VALUE->VALUE;

  elt_to_elts(ARRAY->D,L,U,ELTS) &
 convertVALUEStoELTS(|-VALUES->ELTS') & elts_to_elt(D,L,U,ELTS'->ARRAY')
-----------------------------------------
 |- VALUES,ARRAY->ARRAY';
end modifyP1arrayByElts;
set modifyP1arrayByArray is
judgement |- VALUES,VALUE->VALUE;
 |-values[array(DIM,dim_content(L,U,newELTS))],
 array(D,dim_content(lower LC,UC,oldELTS))
   ->array(D,dim_content(lower LC,UC,newELTS));

 constructOneDimEmpArr(int_const 0 |- LC,UC->zeroARR1)
-----------------------------------------
 |-values[int_const 0],array(D,dim_content(lower LC,upper UC,ELTS))->zeroARR1;

 constructOneDimEmpArr(VALUE |- LC,UC->zeroARR1)
-----------------------------------------
 |-values[VALUE],array(D,dim_content(lower LC,upper UC,ELTS))->zeroARR1;

end modifyP1arrayByArray;
set convertVALUEStoELTS is
judgement |- VALUES -> ELTS;
 |- values[VAL] -> elts[VAL];

 |- VALS -> ELTS
-----------------------------------
 |- values[VAL.VALS] -> elts[VAL.ELTS];
end convertVALUEStoELTS;
set modify_array_by_p12 is
 judgement |- VALUES,VALUES,VALUE->VALUE;

   elt_to_elts(ARRAY->D,L,U,ELTS) & modifyPelts(|-PLIST,L,VALS,ELTS->ELTS')
 & elts_to_elt(D,L,U,ELTS'->ARRAY')
-----------------------------------------------------------------------
 |- PLIST,VALS,ARRAY->ARRAY';
end modify_array_by_p12;
end sisal_array11;
```

A.5 Program Sisal_array_concat

The Typol program *sisal_array_concat.ty* is a collection of sets of rules which handle the array concatenation.

```
import  appendtree(_,_,_) from prolog;
import star(_,_,_),plus(_,_,_),cal_size(_,_,_) from functions ;
import search_in_env(NAME |- ENV : VALUE),
       type_in_env(NAME |- ENV : TYPE_SPEC)
from environment;
import not(|- BOOLEAN -> BOOLEAN)
from sisal_for_exp;
set array_concat is
  judgement SYSTEM, ENV, ENV |- EXPRESSION, EXPRESSION -> VALUES,ENV;
  judgement SYSTEM, ENV, ENV |- VALUE, VALUE => VALUES,ENV;
 elt_to_elts(VALUE1->_,_,_,ELTS1) & elt_to_elts(VALUE2->_,_,_,ELTS2) &
 size_of_whole_array(VALUE1->SIZE1) & size_of_whole_array(VALUE2->SIZE2) &
```

```
 plus(SIZE1,SIZE2,SIZE) & appendtree(ELTS1,ELTS2,ELTS)
 & elts_to_elt(dim 1, 1, upper SIZE, ELTS -> ELT)
 ------------------------------------------------------------------------
 SYSTEM, ENV, SENV |- VALUE1, VALUE2 =>values[ELT],SENV';

 eval_expression(SYSTEM,ENV,SENV|-EXPRESSION1:values[ELT1],SENV') &
 type_in_env(ArrayName2 |- ENV : array_type(_,_))
& search_in_env(ArrayName2|-ENV:ELT2) &
 elt_to_elts(ELT1->_,_,_,ELTS1) & elt_to_elts(ELT2->_,_,_,ELTS2) &
 size_of_whole_array(ELT1->SIZE1) & size_of_whole_array(ELT2->SIZE2) &
 plus(SIZE1,SIZE2,SIZE) & appendtree(ELTS1,ELTS2,ELTS)
& elts_to_elt(dim 1, 1, upper SIZE, ELTS -> ELT)
 ------------------------------------------------------------------------
 SYSTEM, ENV,SENV|- EXPRESSION1, ArrayName2 ->values[ELT],SENV';

 eval_expression(SYSTEM,ENV,SENV|-EXPRESSION1:values[ELT1],SENV') &
 eval_expression(SYSTEM,ENV,SENV'|-EXPRESSION2:values[ELT2],SENV'') &
 elt_to_elts(ELT1->_,_,_,ELTS1) & elt_to_elts(ELT2->_,_,_,ELTS2) &
 size_of_whole_array(ELT1->SIZE1) & size_of_whole_array(ELT2->SIZE2) &
 plus(SIZE1,SIZE2,SIZE) & appendtree(ELTS1,ELTS2,ELTS)
 & elts_to_elt(dim 1, 1, upper SIZE, ELTS -> ELT)
 ------------------------------------------------------------------------
 SYSTEM, ENV,SENV |- EXPRESSION1, EXPRESSION2 ->values[ELT],SENV'';

 type_in_env(ArrayName1 |- ENV : array_type(_,_))
 & type_in_env(ArrayName2 |- ENV : array_type(_,_)) &
 search_in_env(ArrayName1|-ENV:ELT1) & search_in_env(ArrayName2|-ENV:ELT2) &
 elt_to_elts(ELT1->_,_,_,ELTS1) & elt_to_elts(ELT2->_,_,_,ELTS2) &
 size_of_whole_array(ELT1->SIZE1) & size_of_whole_array(ELT2->SIZE2) &
 plus(SIZE1,SIZE2,SIZE) & appendtree(ELTS1,ELTS2,ELTS)
 & elts_to_elt(dim 1, 1, upper SIZE, ELTS -> ELT)
 ------------------------------------------------------------------------
 SYSTEM, ENV,SENV |- ArrayName1, ArrayName2 -> values[ELT],SENV';
end array_concat;
set array_plus_op is
    judgement SYSTEM, ENV, ENV |- EXPRESSION, EXPRESSION -> VALUES, ENV;
 eval_expression(SYSTEM,ENV,SENV|-EXPRESSION1:values[ELT1],SENV') &
 eval_expression(SYSTEM,ENV,SENV'|-EXPRESSION2:values[ELT2],SENV'') &
 array_plus_op2(SYSTEM,ENV|- ELT1,ELT2->VALUES)
 ------------------------------------------------------------------------
 SYSTEM, ENV, SENV |- EXPRESSION1, EXPRESSION2 ->VALUES,SENV'';
end array_plus_op;
set array_plus_op2 is
    judgement SYSTEM, ENV |- VALUE, VALUE -> VALUES;
 size_of_whole_array(ELT1->SAME_SIZE) & size_of_whole_array(ELT2->SAME_SIZE) &
 elt_to_elts(ELT1->dim 1,L,upper U,ELTS1) & elt_to_elts(ELT2->dim 1,_,_, ELTS2) &
 value_list_plus_value_list(SYSTEM, ENV |-ELTS1,ELTS2->ELTS)
 & cal_size(L,U,SIZE) &
 elts_to_elt(dim 1, 1, upper SIZE, ELTS -> ELT)
 ------------------------------------------------------------------------
 SYSTEM, ENV |- ELT1, ELT2 ->values[ELT];

end array_plus_op2;
set array_mult_op is
    judgement SYSTEM, ENV, ENV |- EXPRESSION, EXPRESSION -> VALUES,ENV;
 eval_expression(SYSTEM,ENV,SENV|-EXPRESSION1:values[ELT1],SENV') &
 eval_expression(SYSTEM,ENV,SENV'|-EXPRESSION2:values[SCALER],SENV'') &
 array_mult_op2(SYSTEM, ENV |-ELT1,SCALER->VALUES)
 ------------------------------------------------------------------------
 SYSTEM,ENV,SENV|- EXPRESSION1, EXPRESSION2 ->VALUES,SENV'';
end array_mult_op;
set value_list_plus_value_list is
judgement SYSTEM,ENV|-ELTS,ELTS->ELTS;
SYSTEM,ENV|-elts[],elts[]->elts[];

conversion_type(|- VALUE1, VALUE2 -> LEFT, RIGHT)
& plus(LEFT,RIGHT,VALUE) & SYSTEM,ENV|-ELTS1,ELTS2->ELTS
-------------------------------------------------
```

```
SYSTEM,ENV|-elts[VALUE1.ELTS1],elts[VALUE2.ELTS2]->elts[VALUE.ELTS];
end value_list_plus_value_list;
set not_op_on_elts is
judgement (ELTS->ELTS);
(elts[]->elts[]);

not(|-ELT1->ELT2) & (ELTS1->ELTS2)
-----------------------------------
(elts[ELT1.ELTS1]->elts[ELT2.ELTS2]);
end not_op_on_elts;
end sisal_array_concat;
```

A.6 Program Sisal_array_generate

The Typol program *sisal_array_generate* contains the sets of rules related to the detail of array generation.

```
use sisal;
import plus1(_,_),minus1(_,_)
from functions ;
import diff(_,_) from prolog;
import eval_expression_list(SYSTEM, ENV,ENV |- EXPRESSION_LIST ->  VALUES,ENV),
       eval_expression(SYSTEM, ENV,ENV |- EXPRESSION :  VALUES,ENV)
from sisal_expression;
import plusSTEP(integer,OPT_EXPRESSION:integer)
from sisal_array_reference;
import fill_array(SYSTEM,ENV,ENV,integer,VALUE|- ARRAY_PART_LIST->VALUE)
from sisal_array11;
import modifyOneDimArrByPlaceVals(|-integer,VALUES,VALUE->VALUE)
from sisal_array1;
import classify_selector_part_list(SYSTEM,ENV
 |-SELECTOR_PART_LIST,EXPRESSION_LIST,VALUE->VALUE)
from sisal_array_update;
import type_in_env(NAME |- ENV : TYPE_SPEC),
 type(VALUE -> TYPE_SPEC)
from environment;
set array_generate is
 judgement SYSTEM, ENV,ENV,TYPE_SPEC
|-SIZE_DESCR_LIST, ARRAY_PART_LIST -> VALUES,ENV;

eval_array_part(SYSTEM,ENV|-ARRAY_PART->VALUES)
& translate_values_to_array(SYSTEM,ENV|-TRIPLET2,VALUES->ARRAY)
-------------------------------------------------------
SYSTEM,ENV,SENV,TYPE_SPEC|- size_descr_list[TRIPLET2],
 array_part_list[ARRAY_PART]->values[ARRAY],SENV;

    extract_lower_upper(SYSTEM,ENV|-TRIPLET2->LOWER,UPPER)
  & otherwise(SYSTEM, ENV, TYPE_SPEC|-APL:OWExp)
  & constructOneDimEmpArr(OWExp |-LOWER,UPPER->ARRAY')
  & modifyOneDimArrByAPL(SYSTEM,ENV|-array_part_list[ArrPart.APL],ARRAY'->ARRAY)
  ------------------------------------------------------------------------
  SYSTEM,ENV,SENV,TYPE_SPEC|-size_descr_list[TRIPLET2],
 array_part_list[ArrPart.APL]->values[ARRAY],SENV;

  build_array(SYSTEM,ENV,TYPE_SPEC|-
  size_descr_list[TRIPLET2.SIZE_DESCR_LIST], ARRAY_PART_LIST ->DEF_ARRAY,DIM)
& fill_array(SYSTEM,ENV,SENV,0,DEF_ARRAY|-ARRAY_PART_LIST->ARRAY')
------------------------------------------------------------------------
SYSTEM,ENV,SENV,array_type(_,_)|-
size_descr_list[TRIPLET2.SIZE_DESCR_LIST], ARRAY_PART_LIST->
 values[ARRAY'],SENV;
```

```
  build_array(SYSTEM,ENV,TYPE_SPEC|-
size_descr_list[TRIPLET2.SIZE_DESCR_LIST], ARRAY_PART_LIST ->DEF_ARRAY,DIM)
& fill_array(SYSTEM,ENV,SENV,0,DEF_ARRAY|-ARRAY_PART_LIST->ARRAY')
& change_inside_array_to_content(ARRAY'->ARRAY'')
-------------------------------------------------------------------
SYSTEM,ENV,SENV,TYPE_SPEC|- size_descr_list[TRIPLET2.SIZE_DESCR_LIST],
 ARRAY_PART_LIST-> values[ARRAY''],SENV;
end array_generate;
set change_inside_array_to_content is
 judgement (VALUE -> VALUE);
 flaten_elt_array(ELTS->ELTS')
-------------------------------------------------------------------------
 (array(DIM,dim_content(LOWER,UPPER, ELTS)) ->
array(DIM, dim_content(LOWER,UPPER,ELTS'))) ;
end change_inside_array_to_content;
set flaten_elt_array is
 judgement (ELTS->ELTS);
 array_to_content(ARRAY->DIM_CONTENT)
 ------------------------------------
 (elts[ARRAY]->elts[DIM_CONTENT]);

 array_to_content(ARRAY->DIM_CONTENT) & (ELTS -> ELTS')
-------------------------------------------
 (elts[ARRAY.ELTS]->elts[DIM_CONTENT.ELTS']);
end flaten_elt_array;
set array_to_content is
judgement (VALUE->DIM_CONTENT);
 (array(dim 1,DIM_CONTENT)->DIM_CONTENT);

 flaten_elt_array(ELTS->ELTS')
-----------------------------------------------------------------
 (array(_,dim_content(L,U,ELTS))->dim_content(L,U,ELTS'));
end array_to_content;
set otherwise is
 judgement SYSTEM, ENV, TYPE_SPEC |-ARRAY_PART_LIST:EXPRESSION;
 returnOWExp(SYSTEM,ENV,TYPE_SPEC|- OPT_PLACEMENT, EXPRESSION_LIST -> OWExp)
-------------------------------------------------------------
 SYSTEM,ENV,TYPE_SPEC |-
array_part_list[array_part(OPT_PLACEMENT,EXPRESSION_LIST)]: OWExp;

 SYSTEM, ENV, TYPE_SPEC |-ARRAY_PART_LIST : OWExp
-----------------------------------------------------
 SYSTEM, ENV, TYPE_SPEC |- array_part_list[ARRAY_PART1.ARRAY_PART_LIST]: OWExp;
end otherwise;
set returnOWExp is
judgement SYSTEM,ENV,TYPE_SPEC|- OPT_PLACEMENT,EXPRESSION_LIST->EXPRESSION ;
 SYSTEM,ENV,TYPE_SPEC |- OPT_PLACEMENT,EXPRESSION_LIST->int_const 0;
provided diff(OPT_PLACEMENT,otherwise());

 SYSTEM,ENV,TYPE_SPEC |- otherwise(),expression_list[OWExp]-> OWExp;
end returnOWExp;
set constructOneDimEmpArr is
 judgement EXPRESSION |- integer,integer -> VALUE ;
 judgement EXPRESSION, integer |- integer -> ELTS;
 OWExp, UC |- UC -> elts[OWExp] ;  -- need the terminal condition for a list[]

 plus1(LC',LC'')& OWExp,UC |- LC'' -> ELTS
-------------------------------------
 OWExp,UC |- LC' -> elts[OWExp.ELTS];

 OWExp, UC |- LC -> ELTS
-------------------------------------------------------------------
 OWExp |-LC,UC->array(dim 1,dim_content(lower LC, upper UC, ELTS));
end constructOneDimEmpArr;
set extractP1Vals is
    judgement SYSTEM, ENV |- ARRAY_PART -> integer, VALUES;
   eval_expression_list(SYSTEM,ENV,_|-EXP_LIST->Vals,_)
```

```
   ------------------------------------------------------------
   SYSTEM,ENV|-
array_part(selector(selector_part_list[int_const P1],DSL), EXP_LIST)->P1,Vals;
end extractP1Vals;
set translate_values_to_array is
    judgement SYSTEM, ENV|- TRIPLET2, VALUES -> VALUE;
    judgement |- VALUES -> VALUE;

count_values_to_elements(|-VALUES->ELTS,COUNT)
------------------------------------------------------------
|- VALUES -> array(dim 1, dim_content(lower 1,upper COUNT, ELTS)) ;
end translate_values_to_array;
set build_array is
 judgement SYSTEM,ENV,TYPE_SPEC|-SIZE_DESCR_LIST,ARRAY_PART_LIST
->VALUE,integer ;
   otherwise(SYSTEM, ENV,TYPE_SPEC|- ARRAY_PART_LIST:VALUE)
 & build(SYSTEM,ENV,VALUE|- SIZE_DESCR_LIST -> ARRAY,DIM)
------------------------------------------------------------
  SYSTEM, ENV, TYPE_SPEC |- SIZE_DESCR_LIST,ARRAY_PART_LIST -> ARRAY,DIM;
end build_array;
set build is
 judgement SYSTEM,ENV,EXPRESSION |- SIZE_DESCR_LIST -> VALUE, integer ;
   extract_lower_upper(SYSTEM,ENV|-TRIPLET2->LOWER,UPPER)
 & constructOneDimEmpArr(VALUE|- LOWER,UPPER-> ARR1)
------------------------------------------------------------
SYSTEM, ENV, VALUE |- size_descr_list[TRIPLET2] -> ARR1,1;

extract_lower_upper(SYSTEM,ENV|-TRIPLET2->LOWER,UPPER)
& SYSTEM,ENV,VALUE|- SIZE_DESCR_LIST-> INNER_ARRAY,DIMENSION
& construct_elts_from_inner_array(UPPER |- LOWER, INNER_ARRAY -> ELTS) &
plus1(DIMENSION,DIMENSION')
------------------------------------------------------------------------
SYSTEM, ENV, VALUE |- size_descr_list[TRIPLET2.SIZE_DESCR_LIST]
-> array(dim DIMENSION', dim_content(lower LOWER,
upper UPPER, ELTS)),DIMENSION';
end build;
set construct_elts_from_inner_array is
 judgement integer |- integer,VALUE -> ELTS ;

 UPPER |- UPPER,ARRAY -> elts[ARRAY] ;

 plus1(LOWER',LOWER'') & UPPER |- LOWER'',ARRAY -> ELTS
 ---------------------------------------
 UPPER |- LOWER',ARRAY -> elts[ARRAY.ELTS] ;
end construct_elts_from_inner_array;
set count_values_to_elements is
    judgement |- VALUES -> ELTS, integer;
    |- values[] -> elts[], 0 ;  -- need the terminal condition for a list[]

    |- VALS -> ELTS, COUNT' & plus1(COUNT',COUNT)
    ---------------------------------------
    |- values[VAL1.VALS] -> elts[VAL1.ELTS],COUNT ;
end count_values_to_elements;
set eval_array_part is
    judgement SYSTEM, ENV |- ARRAY_PART -> VALUES;
      eval_expression_list(SYSTEM, ENV,_ |- EXPRESSION_LIST -> values[VALUE],_)
    & type(VALUE->array_type(_,_))
    & apply_placement_to_array(SYSTEM, ENV |- OPT_PLACEMENT,VALUE-> VALUES)
    ------------------------------------------------------------
    SYSTEM, ENV |- array_part(OPT_PLACEMENT,EXPRESSION_LIST) -> VALUES;

    eval_expression_list(SYSTEM, ENV,_ |- EXPRESSION_LIST ->  VALUES,_)
    ------------------------------------------------------------
    SYSTEM, ENV |- array_part(OPT_PLACEMENT,EXPRESSION_LIST) -> VALUES;
end eval_array_part;
set apply_placement_to_array is
 judgement  SYSTEM, ENV |- OPT_PLACEMENT,VALUE-> VALUES;
  modify_array_with_SPL(SYSTEM,ENV|-SPL,ARRAY->ARRAY')
```

```
 & elt_to_elts(ARRAY'->_,_,_,ELTS) & elts_to_values(ELTS->VALUES)
 ------------------------------------------------------------------------------
  SYSTEM, ENV |- selector(SPL,_),ARRAY-> VALUES;
end apply_placement_to_array;
set elts_to_values is
judgement (ELTS->VALUES);
(elts[]->values[]);

 (ELTS->VALUES)
 ----------------------------------
 (elts[ELT.ELTS]->values[ELT.VALUES]);
end elts_to_values;
set modify_array_with_SPL is
judgement SYSTEM,ENV|-SELECTOR_PART_LIST,VALUE->VALUE;
elt_to_elts(ARRAY->D,L,U,ELTS) &
 selectLL_UU_ELTS_to_append(LL,UU,STEP,ELTS->ELTS')
 & elts_to_elt(D,L,U,ELTS'->ARRAY')
-----------------------------------------------------------------
SYSTEM,ENV |-selector_part_list[triplet2(NN,triplet(int_const
LL,int_const UU,STEP))] ,ARRAY->ARRAY';
end modify_array_with_SPL;
set selectLL_UU_ELTS_to_append is
 judgement (integer,integer,OPT_EXPRESSION,ELTS->ELTS);
 selectLL_ELT_from_ELTS(LL,ELTS->ELT)
& plusSTEP(LL,STEP:LL') & (LL',UU,STEP,ELTS->ELTS')
 ------------------------------------------------
 (LL,UU,STEP,ELTS->elts[ELT.ELTS']); provided diff(LL,UU);

 selectLL_ELT_from_ELTS(LL,ELTS->ELT)
 ---------------------------------------------
 (LL,LL,STEP,ELTS->elts[ELT]);
end selectLL_UU_ELTS_to_append;
set selectLL_ELT_from_ELTS is
 judgement (integer,ELTS->ELT);
  (1,elts[ELT.ELTS]->ELT);

  minus1(LL,LL') & (LL',ELTS->ELT)
  --------------------------
  (LL,elts[ELT_DISCARDED.ELTS]->ELT); provided diff(LL,1);
end selectLL_ELT_from_ELTS;
set modifyOneDimArrByAPL is
judgement SYSTEM,ENV|-ARRAY_PART_LIST,VALUE->VALUE;

  extractP1Vals(SYSTEM,ENV|-ARRAY_PART1->Place,Vals)
& modifyOneDimArrByPlaceVals(|-Place,Vals,ARRAY''->ARRAY)
-----------------------------------------------------------------------
SYSTEM,ENV |- array_part_list[ARRAY_PART1],ARRAY''-> ARRAY;

 extractP1Vals(SYSTEM,ENV|-ARRAY_PART1->Place,VALUES1)
 & modifyOneDimArrByPlaceVals(|-Place,VALUES1,ARRAY''->ARRAY')
& SYSTEM,ENV|-ARRAY_PARTS,ARRAY'->ARRAY
-----------------------------------------------------------------------
 SYSTEM,ENV|-array_part_list[ARRAY_PART1.ARRAY_PARTS],ARRAY''-> ARRAY;
end modifyOneDimArrByAPL;
set extract_lower_upper is
 judgement SYSTEM, ENV |- TRIPLET2  -> integer, integer ;

eval_expression(SYSTEM, ENV,_ |- UPPER_EXPRESSION : values[int_const UPPER],_)
-----------------------------------------------------------------------
SYSTEM,ENV |- triplet2(NN,triplet(no_expression(),UPPER_EXPRESSION,NE))
->1, UPPER;

eval_expression(SYSTEM, ENV,_ |- LOWER_EXPRESSION : values[int_const LOWER],_)
& eval_expression(SYSTEM, ENV,_ |- UPPER_EXPRESSION : values[int_const UPPER],_)
-----------------------------------------------------------------------
SYSTEM,ENV |- triplet2(NN,triplet(LOWER_EXPRESSION,UPPER_EXPRESSION,NE))
->LOWER, UPPER;
provided diff(LOWER_EXPRESSION,no_expression());
```

```
end extract_lower_upper;
end sisal_array_generate;
```

A.7 Program Sisal_array_reference

The Typol program *sisal_array_reference* is a collection of rule sets which deal with the detail of the array reference.

```
import eval_expression(SYSTEM, ENV,ENV |- EXPRESSION :  VALUES,ENV)
from sisal_expression;
import search_in_env(NAME |- ENV,ENV : VALUE) from environment;
import getIndexELTS(|-VALUE->ELTS), elts_to_elt(DIM,integer,UPPER,ELTS ->ELT)
from sisal_array1;
set array_reference is
   judgement SYSTEM, ENV,ENV |- EXPRESSION, TRIPLET_OR_EXPRESSION
 -> VALUES,ENV;

 search_in_env(ArrName |- ENV,SENV: Array)
 & ExtractArrElementsBy1Ref(Ref1 |- Array -> arrayElement)
-------------------------------------------------------------
 SYSTEM,ENV,SENV|- ArrName, int_const Ref1 -> values[arrayElement],_;

 search_in_env(ArrName |- ENV,SENV: Array) &
 eval_expression(SYSTEM,ENV,SENV|-
 invocation(FUNCTION_EXP,EXP_LIST) :values[int_const Ref1],_) &
 ExtractArrElementsBy1Ref(Ref1 |- Array -> arrayElement)
-------------------------------------------------------------
 SYSTEM,ENV,SENV |-
 ArrName, invocation(FUNCTION_EXP,EXP_LIST) ->values[arrayElement],_;


 search_in_env(ArrName |- ENV,SENV: Array)
 & eval_expression(SYSTEM, ENV,SENV |- LOWER_EXP : values[int_const Ref1],_)
& eval_expression(SYSTEM, ENV,SENV |- UPPER_EXP : values[int_const Ref2],_)
& ExtractArrElementsBy2Ref(Ref1,Ref2|- Array -> PartialArr)
-------------------------------------------------------------
SYSTEM,ENV,SENV |-
 ArrName, triplet(LOWER_EXP,UPPER_EXP, NE)-> values[PartialArr],_;

 search_in_env(ArrName |- ENV,SENV: Array)
& search_in_env(IndexArrName|-ENV,SENV: IndexArray) &
 getIndexELTS(|-IndexArray->IndexELTS)
& modifyArrByIndexELTS(IndexELTS |- Array -> PartialArr)
-------------------------------------------------------------
 SYSTEM,ENV,SENV|- ArrName, IndexArrName -> values[PartialArr],_;
end array_reference;
set modifyArrByIndexELTS is
judgement ELTS |- VALUE -> VALUE;

modifyELTSByIndexELTS(IndexELTS, 0,LOWER|-ELTS->ELTS',SIZE)
-------------------------------------------------------------
IndexELTS |- array(dim 1,dim_content(lower LOWER,U,ELTS))
-> array(dim 1,dim_content(lower 1,upper SIZE,ELTS'));
end modifyArrByIndexELTS ;
set modifyELTSByIndexELTS is
judgement ELTS,integer,integer|-ELTS->ELTS,integer;

  extractELTbyIndex(Index,LOWER|-ELTS->ELT) & plus1(SIZE,SIZE')
-------------------------------------------------------------
elts[int_const Index],SIZE,LOWER|-ELTS->elts[ELT],SIZE';
```

```
  extractELTbyIndex(Index,LOWER|-ELTS->ELT')
& plus1(SIZE,SIZE') & IndexELTS,SIZE',LOWER|-ELTS->ELTS',SIZE''
-------------------------------------------------------------------------
elts[int_const Index.IndexELTS],SIZE,LOWER|-ELTS->elts[ELT'.ELTS'],SIZE'';
end modifyELTSByIndexELTS;
set extractELTbyIndex is
judgement integer,integer|-ELTS->ELT;
 Index,Index|-elts[this_elt] -> this_elt;
 Index,Index|-elts[this_elt.ELTS] -> this_elt;

  diff(Index,I) & plus1(I,I') & Index,I'|- ELTS -> this_elt
-------------------------------------------------------------------------
 Index,I|-elts[wrong_elt.ELTS] -> this_elt;
end extractELTbyIndex;
set ExtractArrElementsBy1Ref is
judgement integer |- VALUE -> VALUE;
  ExtractArrEltsBy1RefLower(|-Ref1,LOWER,ELTS-> arrayElement)
-------------------------------------------------------------------------
  Ref1 |- array(dim 1, dim_content(lower LOWER,UPPER,ELTS))-> arrayElement;
end ExtractArrElementsBy1Ref;
set ExtractArrElementsBy2Ref is
judgement integer,integer |- VALUE -> VALUE;
   ExtractArrEltsBy2RefLower(|-Ref1,Ref2,LOWER,ELTS-> ELTS')
 & cal_size(Ref1,Ref2,Size)
-------------------------------------------------------------------------
  Ref1,Ref2 |- array(dim 1, dim_content(lower LOWER,UPPER,ELTS))
 -> array(dim 1, dim_content(lower 1,upper Size,ELTS'));
end ExtractArrElementsBy2Ref;
set ExtractArrEltsBy1RefLower is
judgement |- integer, integer, ELTS -> VALUE;
 |- P2,P2,elts[ThisElement]->ThisElement;
 |- P2,P2,elts[ThisElement.ELTS]->ThisElement;

   diff(P2,P2') & plus1(P2',P2'') & |- P2,P2'',ELTS->ThisElement
-------------------------------------------------------------------------
   |- P2,P2',elts[not_this_element.ELTS]->ThisElement;
end ExtractArrEltsBy1RefLower;
set ExtractArrEltsBy2RefLower is
judgement |- integer, integer, integer, ELTS -> ELTS;
  lt(int_const Ref2,int_const P2',true())
-----------------------------------------------------
 |- P2,Ref2,P2',ELTS -> elts[];

   lt(int_const Ref2,int_const P2',false())
& gt(int_const P2,int_const P2',false())
 & plus1(P2',P2'') & |- P2,Ref2,P2'',ELTS ->ELTS'
-------------------------------------------------------------------
 |-P2,Ref2,P2',elts[IncludeThisElement.ELTS]->elts[IncludeThisElement.ELTS'];

   gt(int_const P2,int_const P2',true())
& plus1(P2',P2'') & |-P2,Ref2,P2'',ELTS-> ELTS'
-------------------------------------------------------------------
 |- P2,Ref2,P2',elts[not_this_element.ELTS]->ELTS';
end ExtractArrEltsBy2RefLower;
set eval_array_ref is
judgement SYSTEM,ENV,ENV|-
EXPRESSION,SELECTOR_PART_LIST,DIAG_SPEC_LIST-> VALUES,ENV;
search_in_env(ArrName|-ENV,SENV: Array)
& ExtractP1ArrELT(P1|-Array->P1ArrELT) & elt_to_array(P1ArrELT->P1Arr)
-------------------------------------------------------------------------
SYSTEM,ENV,SENV |-ArrName,selector_part_list[int_const P1,
  triplet2(M,triplet(no_expression(),
no_expression(),no_expression()))], diag_spec_list[]->values[P1Arr],SENV';

   search_in_env(ArrName |- ENV,SENV: Array)
 & ExtractP1ArrELT(P1|-Array->P1ArrELT) & elt_to_array(P1ArrELT->P1Arr)
 & eval_expression(SYSTEM, ENV,SENV |- LOWER_EXP : values[int_const Ref1],_)
 & eval_expression(SYSTEM, ENV,SENV |- UPPER_EXP : values[int_const Ref2],_)
```

```
 & ExtractArrElementsBy2Ref(Ref1,Ref2|- P1Arr -> PartialP1Arr)
-------------------------------------------------------------------------
SYSTEM,ENV,SENV |-ArrName,selector_part_list[int_const P1,
 triplet2(N,triplet(LOWER_EXP,UPPER_EXP,NE))],diag_spec_list[]
->values[PartialP1Arr],SENV';

search_in_env(ArrName |- ENV,SENV: Array)
& eval_expression(SYSTEM, ENV,SENV |- L1 : values[int_const RefL1],_) &
eval_expression(SYSTEM, ENV,SENV |- U1 : values[int_const RefU1],_) &
eval_expression(SYSTEM, ENV,SENV |- L2 : values[int_const RefL2],_) &
eval_expression(SYSTEM, ENV,SENV |- U2 : values[int_const RefU2],_) &
returnIIJJ(SYSTEM,ENV,Array,RefL1,RefU1,RefL2,RefU2|-ISTEP,JSTEP->SIZE,ELTS) &
elts_to_elt(dim 1,1,upper SIZE,ELTS->PartialArr)
-------------------------------------------------------------------------
SYSTEM,ENV,SENV|-ArrName,
selector_part_list[triplet2(II,triplet(L1,U1,ISTEP)),
 triplet2(JJ,triplet(L2,U2,JSTEP))],
                          diag_spec_list[diag_spec(II,name_list[JJ])]
-> values[PartialArr],SENV';

   search_in_env(ArrName |- ENV,SENV: Array) &
 eval_expression(SYSTEM, ENV,SENV |- SP1 : values[int_const P1],_)
 & eval_expression(SYSTEM, ENV,SENV |- SP2 : values[int_const P2],_)
 & ExtractP1ArrELT(P1|-Array->P1ArrELT) &
 elt_to_array(P1ArrELT->P1Arr)
 & ExtractArrElementsBy1Ref(P2|- P1Arr -> PartialP1Arr)
-------------------------------------------------------------------------
SYSTEM,ENV,SENV|-ArrName,selector_part_list[SP1,SP2],
diag_spec_list[]-> values[PartialP1Arr],SENV';
end eval_array_ref;
set returnIIJJ is
judgement SYSTEM,ENV,VALUE,integer,integer,integer,integer
|- OPT_EXPRESSION,OPT_EXPRESSION->integer,ELTS;
  ExtractP1ArrELT(RefU1|-Array->IArrayELT) & elt_to_array(IArrayELT->IArray)
& ExtractArrElementsBy1Ref(JJ |- IArray -> arrayElement)
-------------------------------------------------------------
 SYSTEM,ENV,Array,RefU1,RefU1,JJ,RefU2|-ISTEP,JSTEP->1,elts[arrayElement];

   diff(II,RefU1)   & ExtractP1ArrELT(II|-Array->IArrayELT)
 & elt_to_array(IArrayELT->IArray) & ExtractArrElementsBy1Ref(JJ |- IArray
 -> arrayElement)
 & plusSTEP(II,ISTEP:II') &
 plusSTEP(JJ,JSTEP:JJ') &
SYSTEM,ENV,Array,II',RefU1,JJ',RefU2|- ISTEP,JSTEP->SIZE,ELTS &
 plus1(SIZE,SIZE')
 ------------------------------------------------------------------
 SYSTEM,ENV,Array,II,RefU1,JJ,RefU2|-ISTEP,JSTEP
 ->SIZE',elts[arrayElement.ELTS];
end returnIIJJ;
set elt_to_array is
 judgement (ELT -> VALUE);
 (array(DIM,DIM_CONTENT)->array(DIM,DIM_CONTENT));
 (DIM_CONTENT->array(dim 1,DIM_CONTENT));
end elt_to_array;
set plusSTEP is
judgement (integer, OPT_EXPRESSION: integer);
plus1(II,II')
------------------------
(II,no_expression():II');

plus(II,STEP,II')
------------------------
(II, int_const STEP :II');

minus(II,STEP,II')
-----------------------------------
(II, unary(minus_op(),int_const STEP):II');
end plusSTEP;
```

```
set ExtractP1ArrELT is
judgement integer|-VALUE->ELT;
ExtractP1ELT(P1,LOWER|-ELTS->P1ArrELT)
-------------------------------------------------------
P1|-array(DIM,dim_content(lower LOWER,UPPER,ELTS))->P1ArrELT;
end ExtractP1ArrELT;
set ExtractP1ELT is
judgement integer,integer|-ELTS->ELT;
P1,P1|-elts[ThisIsP1ELT]->ThisIsP1ELT;

P1,P1|-elts[ThisIsP1ELT.ELTS]->ThisIsP1ELT;

  diff(P1,P1') & plus1(P1',P1'') & P1,P1''|-ELTS->ThisIsP1ELT
-------------------------------------------
P1,P1'|-elts[NotThisELT.ELTS]->ThisIsP1ELT;
end ExtractP1ELT;
end sisal_array_reference;
```

A.8 Program Sisal_array_update

The Typol program *sisal_array_update* consists rules which manipulate the detail of the array update.

```
use sisal;

import plus1(_,_),gt(_,_,_),plus(_,_,_),minus(_,_,_),lt(_,_,_)
from functions ;
import diff(_,_) from prolog;
import search_in_env(NAME |- ENV,ENV : VALUE) from environment;
set array_update is
    judgement SYSTEM, ENV,ENV |- EXPRESSION, UPDATE_PART_LIST -> VALUES,ENV;
 search_in_env(NAME |-ENV,SENV: ARRAY)
& update_this_array(SYSTEM,ENV,SENV|-ARRAY,UPDATE_PART_LIST->ARRAY',SENV')
-------------------------------------------------------------------------
SYSTEM,ENV,SENV|- NAME, UPDATE_PART_LIST -> values[ARRAY'],SENV';
end array_update;
set update_this_array is
 judgement SYSTEM,ENV,ENV|- VALUE,UPDATE_PART_LIST->VALUE,ENV;
 update_any_dim_array(SYSTEM,ENV,SENV|- UPDATE_PART,ARRAY->ARRAY',SENV')
-------------------------------------------------------------------------
 SYSTEM,ENV,SENV|- ARRAY, update_part_list[UPDATE_PART] -> ARRAY',SENV';

 update_any_dim_array(SYSTEM,ENV,SENV|- UPDATE_PART,ARRAY->ARRAY',SENV')
 & SYSTEM,ENV,SENV'|- ARRAY', UPDATE_PART_LIST -> ARRAY'',SENV''
-------------------------------------------------------------------------
 SYSTEM,ENV,SENV|-
ARRAY, update_part_list[UPDATE_PART.UPDATE_PART_LIST] -> ARRAY'',SENV'';
end update_this_array;
set update_any_dim_array is
judgement SYSTEM,ENV,ENV|- UPDATE_PART,VALUE->VALUE,ENV;
process_selector(SYSTEM,ENV,SENV
|- SELECTOR, EXPRESSION_LIST, ARRAY ->ARRAY',SENV')
-------------------------------------------------------------------
SYSTEM,ENV,SENV|- update_part(SELECTOR,EXPRESSION_LIST),ARRAY->ARRAY',SENV';
end update_any_dim_array;
set process_selector is
judgement  SYSTEM,ENV,ENV |- SELECTOR, EXPRESSION_LIST, VALUE -> VALUE,ENV;
classify_selector_part_list(SYSTEM,ENV,SENV
|-SELECTOR_PART_LIST,EXPRESSION_LIST,ARRAY->ARRAY',SENV')
-------------------------------------------------------------------------
  SYSTEM,ENV,SENV|- selector(SELECTOR_PART_LIST,diag_spec_list□),
```

```
      EXPRESSION_LIST, ARRAY -> ARRAY',SENV';
end process_selector;
set classify_selector_part_list  is
 judgement SYSTEM,ENV,ENV|-SELECTOR_PART_LIST,EXPRESSION_LIST,VALUE->VALUE,ENV;
eval_expression_list(SYSTEM, ENV,_|-EXPS->VALUES,_)
& update_one_dim_array(SYSTEM,ENV,_|-P,VALUES,ARRAY->ARRAY',_)
-----------------------------------------------------------------
SYSTEM,ENV,SENV|- selector_part_list[int_const P],EXPS,ARRAY->ARRAY',_ ;

  eval_expression_list(SYSTEM, ENV,_ |- EXPS->VALUES,_)
& eval_expression(SYSTEM, ENV,_ |- EXP1:values[int_const E1],_)
& eval_expression(SYSTEM, ENV,_ |- EXP2:values[int_const E2],_)
& update_elts_E1_to_E2(SYSTEM,ENV|- E1,E2,VALUES,ARRAY->ARRAY')
-------------------------------------------------------------------------------
SYSTEM,ENV,_|-
selector_part_list[triplet2(NN,triplet(EXP1,EXP2, no_expression()))],
 EXPS,ARRAY->ARRAY',_;

 eval_expression_list(SYSTEM, ENV,_ |- EXPS->VALUES,_)
& eval_expression(SYSTEM, ENV,_ |- EXP1:values[int_const E1],_)
& eval_expression(SYSTEM, ENV,_ |- EXP2:values[int_const E2],_)
& eval_expression(SYSTEM, ENV,_ |- EXP3:values[int_const STEP],_)
& update_elts_E1_to_E2_with_STEP(SYSTEM,ENV|- E1,E2,STEP,VALUES,ARRAY->ARRAY')
-------------------------------------------------------------------------------
SYSTEM,ENV,_|-
selector_part_list[triplet2(NN,triplet(EXP1, EXP2,EXP3))], EXPS,ARRAY
->ARRAY',_;
provided diff(EXP3,no_expression());

modify_array_with_placement(SYSTEM,ENV,_|- selector(SPL,_),EXPS,ARRAY->ARRAY')
-----------------------------------------------------------------
SYSTEM,ENV,_|- SPL,EXPS,ARRAY->ARRAY',_;

end classify_selector_part_list;
set update_elts_E1_to_E2 is
 judgement SYSTEM,ENV|- integer,integer,VALUES,VALUE->VALUE;
  update_one_dim_array(SYSTEM,ENV,_|-E,VALUES,ARRAY->ARRAY',_)
 ---------------------------------------------
  SYSTEM,ENV|- E,E,VALUES,ARRAY->ARRAY';

   diff(E1,E2)
 & update_one_dim_array(SYSTEM,ENV,_|-E1,values[VALUE],ARRAY->ARRAY',_)
 & plus1(E1,E1') & SYSTEM,ENV|- E1',E2,VALUES,ARRAY'->ARRAY''
 ---------------------------------------------
  SYSTEM,ENV|- E1,E2,values[VALUE.VALUES],ARRAY->ARRAY'';
end update_elts_E1_to_E2;
set update_elts_E1_to_E2_with_STEP is
 judgement SYSTEM,ENV|- integer,integer,integer,VALUES,VALUE->VALUE;
  gt(int_const STEP,0,true()) & gt(int_const E1,int_const E2,true())
 ---------------------------------------------
  SYSTEM,ENV|- E1,E2,STEP,VALUES,ARRAY->ARRAY;

  lt(int_const STEP,0,true()) & lt(int_const E1,int_const E2,true())
 ---------------------------------------------
  SYSTEM,ENV|- E1,E2,STEP,VALUES,ARRAY->ARRAY;

  update_one_dim_array(SYSTEM,ENV,_|-E,VALUES,ARRAY->ARRAY',_)
 ---------------------------------------------
  SYSTEM,ENV|- E,E,STEP,VALUES,ARRAY->ARRAY';

  gt(int_const STEP,0,true()) & lt(int_const E1,int_const E2,true())
 & update_one_dim_array(SYSTEM,ENV,_|-E1,VALUES,ARRAY->ARRAY',_)
 & plus(E1,STEP,E1') & SYSTEM,ENV|- E1',E2,STEP,VALUES,ARRAY'->ARRAY''
 ---------------------------------------------
  SYSTEM,ENV|- E1,E2,STEP,VALUES,ARRAY->ARRAY'';

   lt(int_const STEP,0,true()) & lt(int_const E1,int_const E2,true())
 & update_one_dim_array(SYSTEM,ENV,_|-E1,VALUES,ARRAY->ARRAY',_)
```

```
 & minus(E1,STEP,E1') & SYSTEM,ENV|- E1',E2,STEP,VALUES,ARRAY'->ARRAY''
 ------------------------------------------
  SYSTEM,ENV|- E1,E2,STEP,VALUES,ARRAY->ARRAY'';
end update_elts_E1_to_E2_with_STEP;
set update_one_dim_array is
 judgement SYSTEM,ENV,ENV|-integer,VALUES,VALUE->VALUE,ENV;
 update_elts(|-P,L,VALUES,ELTS->ELTS')
-----------------------------------------------------------------------
 SYSTEM,ENV,_|-P,VALUES,array(DIM,dim_content(lower L,U,ELTS))->
                    array(DIM,dim_content(lower L,U,ELTS')),_;
end update_one_dim_array;
set update_elts is
 judgement |- integer,integer,VALUES,ELTS -> ELTS;
 |- P,P,values[ELT'],elts[ELT] -> elts[ELT'];
 |- P,P,values[ELT'],elts[ELT.ELTS] -> elts[ELT'.ELTS];

   diff(P,I) & plus1(I,I') & |- P,I', VALUES, ELTS -> ELTS'
-----------------------------------------------------------------------
  |- P,I,VALUES,elts[ELT.ELTS] -> elts[ELT.ELTS'];
end update_elts;
set spl_to_p_list is
judgement SYSTEM,ENV |- SELECTOR_PART_LIST -> VALUES;
 SYSTEM,ENV |- selector_part_list[P]-> values[P];

 SYSTEM,ENV |- SPL -> VALS
 ---------------------------------------------------
 SYSTEM,ENV |- selector_part_list[P.SPL] -> values[P.VALS];
end spl_to_p_list;
end sisal_array_generate;
```

A.9 Program Sisal_expression

The Typol program *sisal_expression* contains the most rules. The rules in this program are used to recognize the patterns of the expressions in a Sisal program.

```
use sisal;
use PSP;
set eval_expression_list is
    judgement SYSTEM, ENV, ENV |- EXPRESSION_LIST -> VALUES, ENV;
    _, ENV, SENV |- expression_list[] -> values[],SENV;

    eval_expression(SYSTEM, ENV, SENV |- EXP :  VALUES1, SENV') &
    SYSTEM, ENV, SENV'|- EXPS -> VALUES2, SENV'' &
    appendtree(VALUES1,VALUES2,VALUES)
    ----------------------------------------
    SYSTEM, ENV, SENV |- expression_list[EXP.EXPS] -> VALUES,SENV'';
end eval_expression_list;
set kill_NAME is
judgement (ENV,NAME->ENV);
    (env[],NAME->env[]);

    (ENV,NAME->ENV')
    ----------------------------
    (env[pair(NAME,_).ENV],NAME->ENV');

    (ENV,NAME->ENV')
    ----------------------------
    (env[pair(NAME',VAL).ENV],NAME->env[pair(NAME',VAL).ENV']);
provided diff(NAME',NAME);
end kill_NAME;
set kill_special_variables is
```

```
judgement (ENV,ORDINARY_VARIABLE_LIST->ENV);
    (COMBO,ordinary_variable_list[]->COMBO);

    kill_NAME(COMBO,NAME->COMBO') & (COMBO',OVL->COMBO'')
-------------------------------------------------------
    (COMBO,ordinary_variable_list[NAME.OVL]->COMBO'');
end kill_special_variables;
set substract_env is -- ENV1- ENV2
judgement (ENV,ENV->ENV);
    (ENV1,env[]->ENV1);

    kill_NAME(ENV1,NAME->ENV1') & (ENV1',ENV2->ENV1'')
-------------------------------------------------------
    (ENV1,env[pair(NAME,_).ENV2]->ENV1'');
end substract_env;
set eval_expression is
    judgement SYSTEM, ENV,ENV |- EXPRESSION :  VALUES,ENV;
    var const: CONSTANT;

   liml_execution(SYSTEM, ENV, SENV |- EXP_LIST : VALUES')
   -------------------------------------------------------
   SYSTEM, ENV, SENV |- invocation(name "liml", EXP_LIST) : VALUES',SENV;

   limh_execution(SYSTEM, ENV, SENV |- EXP_LIST : VALUES')
   -------------------------------------------------------
   SYSTEM, ENV, SENV |- invocation(name "limh", EXP_LIST) : VALUES',SENV;

   size_execution(SYSTEM, ENV, SENV |- EXP_LIST : VALUES')
   -------------------------------------------------------
   SYSTEM, ENV, SENV |- invocation(name "size", EXP_LIST) : VALUES',SENV;

   withsubject(SP:function_definition(NAME |- SYSTEM : true(),
               function_def(function_header(NAME, FOR_PARAMS,T),
SPECIAL_VARS, ORDINARY_VARS,EXPS),CALLEE_ADD_SENV,FP))&
 eval_expression_list(SYSTEM,ENV,CURRENT_SENV|-EXP_LIST->VALUES,CURRENT_SENV')
 & bind_parameters(FOR_PARAMS , VALUES -> BIND_PARAMS) &
   appendtree(CALLEE_ADD_SENV,CURRENT_SENV',COMBO_SENV) &
   kill_special_variables(COMBO_SENV,ORDINARY_VARS->COMBO_SENV') &
   withsubject(FP:function_execution(SYSTEM, BIND_PARAMS,COMBO_SENV'
 |- EXPS:VALUES',NEW_COMBO_SENV))
   & substract_env(NEW_COMBO_SENV,CALLEE_ADD_SENV
->COMBO_EXCLUDING_CALLEE_SPECIAL) &
   update_CURRENT_by_NEW_COMBO(CURRENT_SENV',COMBO_EXCLUDING_CALLEE_SPECIAL
 ->NEW_CURRENT_SENV)
   -------------------------------------------------
  SYSTEM,ENV,CURRENT_SENV |-invocation(NAME, EXP_LIST)
 :VALUES',NEW_CURRENT_SENV ; provided system_path(SP) ;

   eval_expression(SYSTEM, ENV, SENV
 |- EXPRESSION: values[closure(assign_function_def
     (function_header(_,FOR_PARAMS,_),FUNC_BODY),
ENV1,SENV1,fun_path(FP))],SENV)&
   eval_expression_list(SYSTEM, ENV, SENV |- EXP_LIST -> VALUES,SENV') &
   bind_parameters(FOR_PARAMS , VALUES -> BIND_PARAMS) &
   appendtree(BIND_PARAMS, ENV1, ENV2) &
   withsubject(FP:eval_expression_list(SYSTEM, ENV2, SENV'
|- FUNC_BODY -> VALS,SENV''))
   ----------------------
   SYSTEM, ENV, SENV |- invocation(EXPRESSION, EXP_LIST) : VALS,SENV'';

  array_update( SYSTEM, ENV, SENV
 |- EXPRESSION, UPDATE_PART_LIST -> VALUES,SENV')
   ---------------------------------------------------------------------
   SYSTEM, ENV, SENV |-
 array_update(EXPRESSION, UPDATE_PART_LIST) :  VALUES,SENV';

array_generate(SYSTEM,ENV,SENV,TYPE_SPEC
|-SIZE_DESCR_LIST, ARRAY_PART_LIST -> VALUES,SENV')
```

```
 ------------------------------------------------------------------
 SYSTEM, ENV, SENV
|- array_gen(TYPE_SPEC, SIZE_DESCR_LIST, ARRAY_PART_LIST) : VALUES,SENV';

type_in_env(ArrayName |- ENV : array_type(_,_)) &
array_reference(SYSTEM, ENV,SENV|- ArrayName, ReferIndex -> VALUES,SENV')
   ------------------------------------------------------------------
  SYSTEM, ENV, SENV
|- stream_ref_or_array_ref(ArrayName, ReferIndex) :  VALUES,SENV';

 eval_array_ref(SYSTEM,ENV,SENV
|-ArrayName,SELECTOR_PART_LIST,DIAG_SPEC_LIST->VALUES,SENV')
------------------------------------------------------------------
 SYSTEM, ENV, SENV |- array_ref(ArrayName,
 selector(SELECTOR_PART_LIST,DIAG_SPEC_LIST)) :  VALUES,SENV';

    eval_expression(SYSTEM,ENV,SENV
|-A1:values[VALUE1],SENV') & type(VALUE1 -> array_type(_,_)) &
    eval_expression(SYSTEM,ENV,SENV'
|-A2:values[VALUE2],SENV'') & type(VALUE2 -> array_type(_,_)) &
    array_concat(SYSTEM, ENV,SENV'' |- VALUE1, VALUE2 => VALUES,NEW_SENV)
    ------------------------------------------------
    SYSTEM, ENV, SENV |- binary(A1,concat_op(),A2) : VALUES,NEW_SENV;

   type_in_env(A |- ENV : array_type(_,_))
 & array_not_op(SYSTEM, ENV,SENV |- A -> VALUES,SENV')
   ------------------------------------------------------------------
   SYSTEM, ENV, SENV |- unary(not_op(),A) : VALUES,SENV';


   type_in_env(EXP1 |- ENV : array_type(_,_))
 & array_mult_op(SYSTEM,ENV,SENV|- EXP1,EXP2->VALUES,SENV')
   ------------------------------------------------
   SYSTEM, ENV, SENV |- binary(EXP1,mult_op(),EXP2):VALUES,SENV';

   SYSTEM, ENV, SENV |- EXP1 :  values[VALUE1],SENV' &
   SYSTEM, ENV, SENV' |- EXP2 : values[VALUE2],SENV''
 & evaluate( |- REL_OP, VALUE1,VALUE2 -> VALUE)
   ------------------------------------------
   SYSTEM, ENV, SENV |- binary(EXP1,REL_OP,EXP2) :  values[VALUE],SENV';

   array_plus_op(SYSTEM,ENV,SENV|- EXP1,EXP2->VALUES,SENV')
   ------------------------------------------------
   SYSTEM, ENV, SENV |- binary(EXP1,plus_op(),EXP2): VALUES,SENV';

   SYSTEM, ENV, SENV |- EXP : values[VALUE], SENV'
& evaluate( |- REL_OP, VALUE -> VALUE1)
   ---------------------------------
   SYSTEM, ENV, SENV |- unary(REL_OP, EXP) : values[VALUE1], SENV';

   eval_let(SYSTEM, ENV, SENV |- let(DECLS,EXPS) ->  VALUES, SENV')
   ---------------------------------------
   SYSTEM, ENV, SENV |- let(DECLS,EXPS) : VALUES, SENV';

  eval_if(SYSTEM, ENV, SENV
|- EXP,EXPS_THEN,ELSEIF_LIST,EXPS_ELSE -> VALUES, SENV')
  ---------------------------------------
  SYSTEM, ENV, SENV
|- conditional(EXP,EXPS_THEN,ELSEIF_LIST,EXPS_ELSE): VALUES, SENV' ;

  eval_case(SYSTEM, ENV, SENV
 |- EXP_CONTROL,CASE_PART_LIST,EXP_LIST -> VALUES,SENV')
  ------------------------------------------------------------
  SYSTEM, ENV, SENV |-
 case(EXP_CONTROL,CASE_PART_LIST,EXP_LIST) :  VALUES,SENV';

   eval_for(SYSTEM, ENV, SENV |- FOR_TOP, FOR_BODY, RETURN_LIST -> VALUES,SENV')
   ------------------------------------------------------
```

```
   SYSTEM, ENV, SENV |- for(FOR_TOP, FOR_BODY, RETURN_LIST) : VALUES,SENV';

   search_in_env(name NAME |- ENV, SENV : VALUE)
   ---------------------------------
   _, ENV, SENV |- name NAME :  values[VALUE],SENV;

   eval_const(ENV |- const -> VALUE)
   ---------------------------------------
   _, ENV, SENV |- const :  values[VALUE],SENV;
end eval_expression;
set update_CURRENT_by_NEW_COMBO is
judgement  (ENV,ENV->ENV);
(env[],NEW_COMBO->env[]);
search_in_env0(name NAME |- NEW_COMBO : NEW_VALUE,1)
-----------------------------------------------
(env[pair(name NAME,VALUE)],NEW_COMBO->env[pair(name NAME,NEW_VALUE)]);

search_in_env0(name NAME |- NEW_COMBO : _,0)
-----------------------------------------------
(env[pair(name NAME,VALUE)],NEW_COMBO->env[pair(name NAME,VALUE)]);

search_in_env0(name NAME |- NEW_COMBO : NEW_VALUE,1)
 & (CURRENT,NEW_COMBO->CURRENT')
-----------------------------------------------
(env[pair(name NAME,VALUE).CURRENT],NEW_COMBO
->env[pair(name NAME,NEW_VALUE).CURRENT']);

search_in_env0(name NAME |- NEW_COMBO :_,0) & (CURRENT,NEW_COMBO->CURRENT')
-----------------------------------------------
(env[pair(name NAME,VALUE).CURRENT],NEW_COMBO
 ->env[pair(name NAME,VALUE).CURRENT']);
end update_CURRENT_by_NEW_COMBO;
set eval_let is
    judgement SYSTEM, ENV, ENV |- EXPRESSION -> VALUES,ENV;
    declarations(SYSTEM, ENV, SENV |- DECL_DEF_LIST : ENV1, SENV1) &
    eval_expression_list(SYSTEM, ENV1, SENV1 |- EXP_LIST -> VALUES,SENV')
    -------------------------------------------------
    SYSTEM, ENV, SENV |- let(DECL_DEF_LIST, EXP_LIST) ->  VALUES,SENV';
end eval_let;
set declarations is
    judgement SYSTEM, ENV, ENV |- DECL_DEF_LIST : ENV, ENV;
    judgement SYSTEM, ENV, ENV |- DECL_DEF -> ENV, ENV;
   _, ENV, SENV |- decl_def_list[] : ENV, SENV ;

 SYSTEM, ENV, SENV |- DECL_DEF -> ENV1, SENV1
 & SYSTEM, ENV1, SENV1 |- DECL_DEFS : ENV2, SENV2
 ------------------------------------------------------------
 SYSTEM, ENV, SENV |- decl_def_list[DECL_DEF.DECL_DEFS] : ENV2, SENV2 ;

   collect_decl_ids(DECL_LIST -> DECL_ID_LIST)
& eval_expression_list(SYSTEM, ENV, SENV |- EXP_LIST -> VALUES,SENV') &
   assignments2(SENV'|- DECL_ID_LIST, VALUES : ENV1, SENV1)
& appendtree(ENV1, ENV, ENV2)
   ---------------------------------
   SYSTEM, ENV, SENV |- decl_def(DECL_LIST, EXP_LIST) -> ENV2, SENV1;
end declarations;
set evaluate_or_not is
    judgement SYSTEM, ENV, ENV, DECL_ID_LIST |- EXPRESSION_LIST => VALUES,ENV;
    judgement SYSTEM, ENV, ENV, DECL_ID |- EXPRESSION_LIST = VALUES,ENV;

    SYSTEM, ENV, SENV, DECL_ID |- EXPS = VALUES, SENV'
    ------------
    SYSTEM, ENV, SENV, decl_id_list[DECL_ID] |- EXPS => VALUES, SENV';

    SYSTEM, ENV, SENV , DECL_ID |- expression_list[EXP1] = VALS1, SENV' &
    SYSTEM, ENV, SENV , DECL_IDS |- EXPS => VALS2, SENV'' &
    appendtree(VALS1, VALS2, VALS)
    ----------
```

```
    SYSTEM, ENV, SENV ,
 decl_id_list[DECL_ID.DECL_IDS] |- expression_list[EXP1.EXPS] => VALS, SENV'';

    eval_expression_list(SYSTEM, ENV,SENV |- EXPS -> VALUES,SENV')
    ----------
    SYSTEM, ENV, SENV , name NAME |- EXPS = VALUES,SENV' ;

    SYSTEM, ENV, SENV , function_header(NAME,PARAMS,TYPES) |- EXPS =
         values[closure(assign_function_def(function_header(NAME,PARAMS,TYPES),
                        EXPS), ENV, SENV, fun_path(subject))],SENV;
end evaluate_or_not ;
end sisal_expression;
```

A.10 Program Sisal_for_exp

The Typol program *sisal_for_exp.ty* contains rule sets which deal with the "for" expressions.

```
....
set eval_for is
   judgement SYSTEM, ENV, ENV |- FOR_TOP, FOR_BODY, RETURN_LIST -> VALUES,ENV;
   construct_default_values_bottom(|- RETURN_LIST -> DEFAULT_VALUES_BOTTOM) &
   iterate_body(SYSTEM, ENV, SENV, DEFAULT_VALUES_BOTTOM
|- no_expression(),FOR_BODY, RETURN_LIST -> _, _, VALUES)
   ------------------------------------------------
   SYSTEM, ENV, SENV |- no_for_top(), FOR_BODY, RETURN_LIST -> VALUES,SENV;
  construct_default_values_bottom(|- RETURN_LIST -> DEFAULT_VALUES_BOTTOM) &
  declarations(SYSTEM, ENV, SENV |- DDL : ENV1, SENV1) &
  iterate_body(SYSTEM, ENV1, SENV1, DEFAULT_VALUES_BOTTOM
|- TEST_FOR, FOR_BODY, RETURN_LIST -> _, _, VALUES)
  ------------------------------------
  SYSTEM, ENV, SENV
  |- for_top(dot_in_list[],DDL,TEST_FOR),FOR_BODY, RETURN_LIST -> VALUES,SENV1;

 declarations(SYSTEM, ENV,SENV |- DDL : ENV1, SENV1) &
 construct_default_values_bottom(|- RETURN_LIST -> DEFAULT_VALUES_BOTTOM) &
 eval_expression(SYSTEM,ENV,SENV |- ARRAY_NAME:values[ARRAY],SENV) &
 elt_to_elts(ARRAY-> _,_,_,ELTS) &
 iterate_body_with_Element(SYSTEM,ENV1,SENV1,DEFAULT_VALUES_BOTTOM,E,ELTS|-
                     FOR_BODY,RETURN_LIST ->_, SENV2, VALUES)
-------------------------------------------------------------------------------
SYSTEM, ENV, SENV|- for_top(dot_in_list[loop_range(E,ARRAY_NAME,index_list[])],
 DDL,no_expression()), FOR_BODY, RETURN_LIST -> VALUES,SENV2;

 declarations(SYSTEM, ENV,SENV |- DDL : ENV1,SENV1) &
 construct_default_values_bottom(|- RETURN_LIST -> DEFAULT_VALUES_BOTTOM) &
 eval_expression(SYSTEM,ENV,SENV|-L_EXP:values[int_const L],SENV)&
 eval_expression(SYSTEM,ENV,SENV|-U_EXP:values[int_const U],SENV)&
 eval_expression(SYSTEM,ENV,SENV|-L_EXP2:values[int_const L2],SENV)&
 eval_expression(SYSTEM,ENV,SENV|-U_EXP2:values[int_const U2],SENV)&
 iterate_body_with_index2(SYSTEM,ENV1,SENV1,DEFAULT_VALUES_BOTTOM,
   I,I2,L,L2,U,U2 |-FOR_BODY,RETURN_LIST ->_,VALUES)
-------------------------------------------------------------------------------
SYSTEM, ENV,SENV|-
for_top(dot_in_list[ loop_range(I,triplet(L_EXP,U_EXP,no_expression()),
 index_list[]),
loop_range(I2,triplet(L_EXP2,U_EXP2,no_expression()), index_list[])],
DDL,no_expression()),FOR_BODY, RETURN_LIST -> VALUES,SENV1;

 declarations(SYSTEM, ENV,SENV |- DDL : ENV1, SENV1) &
 construct_default_values_bottom(|- RETURN_LIST -> DEFAULT_VALUES_BOTTOM) &
```

```
 eval_expression(SYSTEM,ENV,SENV|-L_EXP:values[int_const L],SENV)&
 eval_expression(SYSTEM,ENV,SENV|-U_EXP:values[int_const U],SENV)&
 iterate_body_with_cross(SYSTEM,ENV1,SENV,
   DEFAULT_VALUES_BOTTOM,I,L,U,I2,L2_EXP,U2_EXP
   |-FOR_BODY,RETURN_LIST ->ENV9,VALUES)
-------------------------------------------------------------------------------
SYSTEM, ENV,SENV|-
for_top(cross_in_list[loop_range(I,
triplet(L_EXP,U_EXP,no_expression()), index_list[]),
loop_range(I2,triplet(L2_EXP,U2_EXP,no_expression()), index_list[])],
DDL,no_expression()),FOR_BODY, RETURN_LIST -> VALUES,SENV1;
end eval_for;
set iterate_body is
   judgement SYSTEM, ENV, ENV, VALUES |-
  TEST, FOR_BODY, RETURN_LIST -> ENV, ENV, VALUES;

    execute_body(SYSTEM, ENV,SENV |- DDL : ENV1, SENV1) &
    eval_return(SYSTEM, ENV1,SENV1, OLD_RETURN |- RETURN_LIST -> NEW_RETURN)&
    return_for(SYSTEM, ENV1,SENV1, NEW_RETURN |- RETURN_LIST -> VALUES)
    ---------------------------------------------
    SYSTEM, ENV,SENV, OLD_RETURN
|- no_expression(),for_body(DDL, no_expression()), RETURN_LIST
         -> ENV1, SENV1, VALUES;

    eval_test(SYSTEM, ENV, SENV |- PRETEST -> true()) &
    execute_body(SYSTEM, ENV, SENV |- DDL : ENV1, SENV1) &
    eval_test(SYSTEM, ENV1, SENV1 |- TEST -> true()) &
    eval_return(SYSTEM, ENV1, SENV1, OLD_RETURN |- RETURN_LIST -> NEW_RETURN)&
    iterate_body(SYSTEM, ENV1, SENV1, NEW_RETURN
|- PRETEST,for_body(DDL, TEST), RETURN_LIST -> ENV2, SENV2, VALUES)
    ---------------------------------------------
    SYSTEM, ENV, SENV,OLD_RETURN
|- PRETEST,for_body(DDL,TEST), RETURN_LIST -> ENV2, SENV2, VALUES;

    eval_test(SYSTEM, ENV, SENV |- PRETEST ->  true()) &
    execute_body(SYSTEM, ENV, SENV |- DDL : ENV1, SENV1) &
    eval_test(SYSTEM, ENV1, SENV1 |- TEST ->  false()) &
    eval_return(SYSTEM, ENV1, SENV1, OLD_RETURN |- RETURN_LIST -> NEW_RETURN)&
    return_for(SYSTEM, ENV1, SENV1, NEW_RETURN |- RETURN_LIST -> VALUES)
    ---------------------------------------------
    SYSTEM, ENV, SENV, OLD_RETURN
 |- PRETEST, for_body(DDL, TEST), RETURN_LIST -> ENV1,SENV1, VALUES;

    eval_test(SYSTEM, ENV, SENV |- PRETEST ->  false()) &
    return_for(SYSTEM, ENV, SENV, OLD_RETURN |- RETURN_LIST -> VALUES)
    ---------------------------------------------
    SYSTEM, ENV, SENV, OLD_RETURN
|- PRETEST, for_body(DDL, TEST), RETURN_LIST -> ENV, SENV, VALUES;
end iterate_body;
set iterate_body2 is
   judgement SYSTEM, ENV, ENV, VALUES|- FOR_BODY, RETURN_LIST ->VALUES,ENV,ENV;
    execute_body(SYSTEM,ENV,SENV |- DDL : ENV1, SENV1) &
    eval_return2(SYSTEM,ENV1,SENV1,OLD_VALUES
|- RETURN_LIST->NEW_VALUES,NEW_SENV)
    ---------------------------------------------------------------------
   SYSTEM,ENV,SENV,OLD_VALUES|-
 for_body(DDL,no_expression()),RETURN_LIST -> NEW_VALUES,ENV1,NEW_SENV;
end iterate_body2;
set iterate_body_with_Element is
   judgement SYSTEM, ENV,ENV, VALUES, NAME, ELTS |- FOR_BODY, RETURN_LIST ->
      ENV, ENV, VALUES;

   SYSTEM,ENV,SENV,OLD_VALUES,_,elts[] |- for_body(DDL,TEST), _ -> ENV,
      SENV, OLD_VALUES;

   assignments2(SENV |- decl_id_list[E],values[ELT]: ENVI,SENVI) &
   appendtree(ENV,ENVI,ENV') &
   appendtree(SENV,SENVI,SENV') &
```

```
   iterate_body2(SYSTEM,ENV',SENV',OLD_VALUES
 |- for_body(DDL,TEST),RETURN_LIST->NEW_VALUES,ENV9,NEW_SENV) &
   SYSTEM,ENV,NEW_SENV,NEW_VALUES,E,ELTS
|- for_body(DDL,TEST),RETURN_LIST -> ENV1,SENV_FINAL, VALUES
   ------------------------------------------------------------------------
   SYSTEM,ENV,SENV,OLD_VALUES,E,elts[ELT.ELTS]
 |- for_body(DDL,TEST), RETURN_LIST -> ENV1,SENV_FINAL,VALUES;
end iterate_body_with_Element;
set iterate_body_with_index2 is
 judgement SYSTEM, ENV, ENV, VALUES, NAME, NAME, integer, integer,
     integer, integer |- FOR_BODY, RETURN_LIST -> ENV, VALUES;

  assignments2(SENV|- decl_id_list[I],values[int_const L]: ENVI,SENVI) &
  appendtree(ENV,ENVI,ENV') &
  assignments2(SENV|- decl_id_list[I2],values[int_const L2]: ENVI2,SENVI2) &
  appendtree(ENV',ENVI2,ENV'') &
  iterate_body2(SYSTEM, ENV'',SENV,OLD_VALUES
|- for_body(DDL,TEST),RETURN_LIST->NEW_VALUES,ENV9,NEW_SENV) &
  plus1(L,L') & plus1(L2,L2') &
  SYSTEM,ENV,SENV,NEW_VALUES,I,I2,L',L2',U,U2
|-for_body(DDL,TEST),RETURN_LIST ->ENV1,VALUES
  ------------------------------------------------------------------------
  SYSTEM,ENV,SENV,OLD_VALUES,I,I2,L,L2,U,U2
|- for_body(DDL,TEST), RETURN_LIST -> ENV1, VALUES; provided diff(L,U);

   assignments2(SENV |- decl_id_list[I],values[int_const L]: ENVI,SENVI) &
   appendtree(ENV,ENVI,ENV') &
   assignments2(SENV |- decl_id_list[I2],values[int_const L2]: ENVII,SENVII) &
   appendtree(ENV',ENVII,ENV'') &
   iterate_body2(SYSTEM, ENV'',SENV,OLD_VALUES|-for_body(DDL,TEST),RETURN_LIST
      ->FINAL_VALUES,ENV9,NEW_SENV) &
   handle_reduction_or_array_of(SYSTEM,ENV,FINAL_VALUES
 |- RETURN_LIST->FINAL_VALUES2)
 ------------------------------------------------------------------
  SYSTEM,ENV,SENV,OLD_VALUES,I,I2,L,L2,L,L2
|- for_body(DDL,TEST),RETURN_LIST->ENV1,FINAL_VALUES2;
end iterate_body_with_index2;
set iterate_body_with_cross is
judgement SYSTEM, ENV, ENV, VALUES, NAME, integer, integer, NAME, EXPRESSION,
      EXPRESSION |- FOR_BODY, RETURN_LIST -> ENV, VALUES;
judgement SYSTEM, ENV, ENV, VALUES, NAME, integer, integer, NAME, EXPRESSION,
      integer, integer |- FOR_BODY, RETURN_LIST :ENV, VALUES;
assignments(|- decl_id_list[I1],values[int_const L1]: ENVI1) &
appendtree(ENV,ENVI1,ENV') &
eval_expression(SYSTEM,ENV',SENV|-L2_EXP:values[int_const L2],SENV)&
eval_expression(SYSTEM,ENV',SENV|-U2_EXP:values[int_const U2],SENV)&
SYSTEM,ENV,SENV,OLD_VALUES,I1,L1,U1,I2,L2_EXP,L2,U2
 |-for_body(DDL,TEST),RETURN_LIST:ENV1, VALUES
------------------------------------------------------------------------
 SYSTEM,ENV,SENV,OLD_VALUES,I1,L1,U1,I2,L2_EXP,U2_EXP
|- for_body(DDL,TEST), RETURN_LIST -> ENV1, VALUES;

 assignments(|- decl_id_list[I1],values[int_const L1]: ENVI1) &
 appendtree(ENV,ENVI1,ENV') &
 assignments(|- decl_id_list[I2],values[int_const L2]: ENVI2) &
 appendtree(ENV',ENVI2,ENV'') &
 iterate_body2(SYSTEM, ENV'',SENV,OLD_VALUES
|- for_body(DDL,TEST),RETURN_LIST->NEW_VALUES,ENV9,NEW_SENV) &
 plus1(L2,L2') &
 SYSTEM,ENV,SENV,NEW_VALUES,I1,L1,U1,I2,L2_EXP,L2',U2
 |-for_body(DDL,TEST),RETURN_LIST: ENV1,VALUES
 ------------------------------------------------------------------------
 SYSTEM,ENV,SENV,OLD_VALUES,I1,L1,U1,I2,L2_EXP,L2,U2
|- for_body(DDL,TEST), RETURN_LIST: ENV1, VALUES;
 provided diff(L2,U2);

   assignments( |- decl_id_list[I1],values[int_const L1]: ENVI1) &
   appendtree(ENV,ENVI1,ENV') &
```

```
   assignments( |- decl_id_list[I2],values[int_const U2]: ENVI2) &
   appendtree(ENV',ENVI2,ENV'') &
   iterate_body2(SYSTEM,ENV'',SENV,OLD_VALUES
|-for_body(DDL,TEST),RETURN_LIST->NEW_VALUES,ENV9,NEW_SENV) &
   plus1(L1,L1') &
   assignments( |- decl_id_list[I1],values[int_const L1']: ENVI1') &
   appendtree(ENV,ENVI1',ENV3) &
   eval_expression(SYSTEM,ENV3,SENV |-L2_EXP:values[int_const L2],SENV)&
  SYSTEM,ENV,SENV,NEW_VALUES,I1,L1',U1,I2,L2_EXP,L2,U2
|-for_body(DDL,TEST),RETURN_LIST : ENV1,FINAL_VALUES2
 ------------------------------------------------------------------
  SYSTEM,ENV,SENV,OLD_VALUES,I1,L1,U1,I2,L2_EXP,U2,U2
|- for_body(DDL,TEST),RETURN_LIST: ENV1,FINAL_VALUES2;
  provided diff(L1,U1);

   assignments(|- decl_id_list[I1],values[int_const U1]: ENVI1) &
   appendtree(ENV,ENVI1,ENV') &
   assignments(|- decl_id_list[I2],values[int_const U2]: ENVI2) &
   appendtree(ENV',ENVI2,ENV'') &
   iterate_body2(SYSTEM, ENV'',SENV,OLD_VALUES|-for_body(DDL,TEST),RETURN_LIST
      ->FINAL_VALUES,ENV9,NEW_SENV) &
   handle_reduction_or_array_of(SYSTEM,ENV,FINAL_VALUES
 |- RETURN_LIST->FINAL_VALUES2)
 ------------------------------------------------------------------
  SYSTEM,ENV,SENV,OLD_VALUES,I1,U1,U1,I2,L2_EXP,U2,U2
|- for_body(DDL,TEST),RETURN_LIST: ENV1,FINAL_VALUES2;
end iterate_body_with_cross;
set return_for is
    judgement SYSTEM, ENV, ENV, VALUES |- RETURN_LIST -> VALUES;
    judgement SYSTEM, ENV, ENV, VALUE |- RETURN : VALUE;

    SYSTEM, ENV, SENV, OLD_VAL |- RETURN : NEW_VAL
    --------------------------------
    SYSTEM, ENV, SENV, values[OLD_VAL]
|- return_list[RETURN] -> values[NEW_VAL];

    SYSTEM, ENV, SENV, OLD_VAL |- RETURN : NEW_VAL &
    SYSTEM, ENV, SENV, VALS |- RETURNS -> NEW_VALS
    --------------------------------
    SYSTEM, ENV, SENV, values[OLD_VAL.VALS]
|- return_list[RETURN.RETURNS] -> values[NEW_VAL.NEW_VALS];

    SYSTEM, ENV, SENV, OLD_VAL |- return_stream(EXP,F) : OLD_VAL;
    SYSTEM, ENV, SENV, OLD_VAL |- return_value(Exp,F) : OLD_VAL;

    plus_list(|- LIST_VAL -> SOMME)
    --------------------------------
    SYSTEM, ENV, SENV, LIST_VAL
|- return_value(invocation(name "sum",PARAMS),_) : SOMME;

end return_for;
set construct_default_values_bottom is
    judgement |- RETURN_LIST -> VALUES;
    judgement |- RETURN : VALUE;

    |- RETURN : VAL
    ------------------------
    |- return_list[RETURN] -> values[VAL];

    |- RETURN : VAL &
    |- RETURNS -> VALS
    ------------------------
    |- return_list[RETURN.RETURNS] -> values[VAL.VALS];

    |- return_value(Exp,_) : int_const 0; provided diff(Exp, invocation(_,_));
    |- return_array(_,_,_): array(dim 0,dim_content(lower 1,upper 0,elts[]));
end construct_default_values_bottom;
set eval_return is
```

```
    judgement SYSTEM, ENV, ENV, VALUES |- RETURN_LIST -> VALUES;
    judgement SYSTEM, ENV, ENV, VALUE |- RETURN : VALUE;

   SYSTEM, ENV, SENV, OLD_VAL
|- RETURN : NEW_VAL & SYSTEM, ENV, SENV, VALS |- RETURNS -> NEW_VALS
    --------------------------------
    SYSTEM, ENV, SENV, values[OLD_VAL.VALS]
|- return_list[RETURN.RETURNS] -> values[NEW_VAL.NEW_VALS];

    eval_filter(SYSTEM, ENV, SENV
|- F -> true()) & eval_expression(SYSTEM, ENV, SENV
|- EXP : values[VAL],SENV) &
    appendtree(OLD_VAL, stream[VAL], NEW_STREAM)
    --------------------------------
    SYSTEM, ENV, SENV, OLD_VAL |- return_stream(EXP,F) : NEW_STREAM ;

    eval_filter(SYSTEM, ENV, SENV |- F -> false())
    --------------------------------
    SYSTEM, ENV, SENV, OLD_VAL |- return_stream(EXP,F) : OLD_VAL;

    eval_filter(SYSTEM, ENV, SENV |- F -> false())
    --------------------------------
    SYSTEM, ENV, SENV, OLD_VAL |- return_value(EXP,F) : OLD_VAL;

    eval_filter(SYSTEM, ENV, SENV |- F -> true()) &
    eval_expression(SYSTEM, ENV, SENV |- EXP : values[NEW_VAL],SENV)
    --------------------------------
    SYSTEM, ENV, SENV, OLD_VAL |- return_value(EXP,F) : NEW_VAL;

    eval_filter(SYSTEM, ENV, SENV |- F -> true()) &
    eval_expression_list(SYSTEM, ENV, SENV |- PARAMS -> values[VALUE],SENV)&
    appendtree(stream[VALUE],OLD_VAL,NEW_VAL)
    ------------------------------------------------
    SYSTEM, ENV, SENV, OLD_VAL
|- return_value(invocation(name "sum",PARAMS),F) : NEW_VAL;

    eval_filter(SYSTEM, ENV,SENV |- F -> true()) &
    eval_expression_list(SYSTEM, ENV, SENV |- PARAMS -> values[VALUE],SENV)&
    appendtree(stream[VALUE],OLD_VAL,NEW_VAL)
    --------------------------------
    SYSTEM, ENV, SENV, OLD_VAL
|- return_value(invocation(name "product",PARAMS),F) : NEW_VAL;

    eval_filter(SYSTEM, ENV, SENV |-F -> true()) &
    eval_expression_list(SYSTEM, ENV,SENV |- PARAMS -> values[VALUE],SENV)  &
    appendtree(stream[VALUE],OLD_VAL,NEW_VAL)
    ----------------
    SYSTEM, ENV, SENV, OLD_VAL
|-return_value(invocation(name "minval", PARAMS),F) : NEW_VAL;

    eval_filter(SYSTEM, ENV, SENV |-F -> true()) &
    eval_expression_list(SYSTEM, ENV ,SENV|- PARAMS ->values[VALUE],SENV)  &
    appendtree(stream[VALUE],OLD_VAL,NEW_VAL)
    ----------------
    SYSTEM, ENV, SENV, OLD_VAL
|-return_value(invocation(name "maxval", PARAMS),F) : NEW_VAL;
end eval_return;
set execute_body is
    judgement SYSTEM, ENV, ENV |- DECL_DEF_LIST : ENV, ENV;
   declarations(SYSTEM, ENV, SENV |- DDL : ENV1, SENV1)
   --------
   SYSTEM, ENV, SENV |- DDL : ENV1, SENV1;
end execute_body;
set elt_to_elts is
 judgement ( ELT -> DIM,integer,UPPER,ELTS);
 (array(D,dim_content(lower LC,U,ELTS)) -> D,LC,U,ELTS);
end elt_to_elts;
end sisal_for_exp;
```

A.11 Program System_definition

The Typol program *system_definition.ty* is a collection of rules to map the Sisal program body to Typol variables.

```
use sisal;
use PSP;
    import diff(_,_), appendtree(_,_,_) from prolog;
set function_definition is
judgement NAME |- sisal : BOOLEAN, FUNCTION_DEF, ENV, PATH ;
......

init_special_variable(SPECIAL_VAR_LIST->LSENV)
-------------------------------------------------------------------------------
NAME |- function_def(function_header(NAME,P,T),
                SPECIAL_VAR_LIST,ORDINARY_VAR_LIST,EXP_LIST): true(),
        function_def(function_header(NAME,P,T),
              SPECIAL_VAR_LIST,ORDINARY_VAR_LIST,EXP_LIST),
 LSENV, s(subject, int 2);

NAME |- function_def(function_header(NAME,P,T), special_variable_list[],
 ordinary_variable_list[],EXP_LIST): true(),
 function_def(function_header(NAME,P,T),
               SPECIAL_VAR_LIST,ORDINARY_VAR_LIST, EXP_LIST),
  env[], s(subject, int 2);

NAME |- function_def(function_header(NAME',_,_),_,_, _) : false(),_,_,_;
provided diff(NAME, NAME');
end function_definition ;
set init_special_variable is
judgement (SPECIAL_VARIABLE_LIST->ENV);
(special_variable_list[]->env[]);

(SVL->SENV)
-----------------------------------------------
(special_variable_list[S.SVL]->env[pair(S,int_const 0).SENV]);
end init_special_variable;
end system_definition;
```

Appendix B

The Metal File

The Metal file is used to specify the syntax of a language. The contents of the Metal program, "sisal.metal", which is related to specify the syntax of extended-Sisal are:

```
definition of sisal is
...

 chapter 'Functions'
     rules
<Int_Function_Def> ::= "function" <Name> "(" <Opt_Int_Param_Decls>
 <Opt_Return> ")" <Opt_Semicolon> ;
int_function_def(<Name>, <Opt_Int_Param_Decls>, <Opt_Return>)

<Opt_Int_Param_Decls> ::= ;
int_param_decl_list-list(())

<Opt_Int_Param_Decls> ::= <Int_Param_Decls> ;
<Int_Param_Decls>

<Int_Param_Decls> ::= <Int_Param_Decl> ;
int_param_decl_list-list((<Int_Param_Decl>))

<Int_Param_Decls> ::= <Int_Param_Decls> "," <Int_Param_Decl> ;
int_param_decl_list-post(<Int_Param_Decls>, <Int_Param_Decl>)

<Int_Param_Decl> ::= <Mode> <Type_Spec> ;
int_param_decl(<Mode>, <Type_Spec>)

<Mode> ::= "in" ;
in_mode()

<Mode> ::= "out" ;
out_mode()

<Mode> ::= "inout" ;
inout_mode()

<Opt_Return> ::= ;
type_spec_list-list(())

<Opt_Return> ::= "returns" <Type_Spec_List> ;
<Type_Spec_List>

<Function_Def> ::= "forward" "function" <Name> "("
   <Opt_Type_Spec_List> "returns" <Type_Spec_List> ")"
   <Opt_Semicolon>;
```

```
forward_function_def(<Name>,
<Opt_Type_Spec_List>, <Type_Spec_List>)

-- YSC 3/30/95 comment the below, because I add the special variable.
--  <Function_Def> ::= "function" <Function_Header> <Expression_List>
--    "end" "function" ;
-- function_def(<Function_Header>, <Expression_List>)
-- YSC 3/30/95 insert "special" in the function definition abstract syntax
-- YSC add the following rule for special function extention 2/8/96
     <Function_Def> ::= "special" "function" <Function_Header>
      <Opt_Special_variable_List> <Opt_Ordinary_variable_List> <Expression_List>
   "end" "function" ;
       function_def(<Function_Header>,
<Opt_Special_variable_List>, <Opt_Ordinary_variable_List>,  <Expression_List>)

     <Function_Def> ::= "function" <Function_Header>
      <Opt_Special_variable_List> <Opt_Ordinary_variable_List> <Expression_List>
   "end" "function" ;
        function_def(<Function_Header>,
<Opt_Special_variable_List>, <Opt_Ordinary_variable_List>,  <Expression_List>)


<Function_Header> ::= <Name> "(" "returns" <Type_Spec_List> ")" ;
function_header(<Name>, param_decl_list-list(()),
<Type_Spec_List>)

<Function_Header> ::= <Name> "(" <Param_Decls> <Opt_Comma> "returns"
<Type_Spec_List> ")" ;
function_header(<Name>, <Param_Decls>, <Type_Spec_List>)

<Param_Decls> ::= <Param_Decl>;
param_decl_list-list((<Param_Decl>))

<Param_Decls> ::= <Param_Decls> "," <Param_Decl>  ;
param_decl_list-post(<Param_Decls>, <Param_Decl>)

<Param_Decl> ::= <Name_List> ":" <Type_Spec> ;
param_decl(<Name_List>, <Type_Spec>)

<Opt_Comma> ::= ;
null()

<Opt_Comma> ::= "," ;
null ()

    abstract syntax
int_function_def -> NAME INT_PARAM_DECL_LIST TYPE_SPEC_LIST;
INT_PARAM_DECL_LIST ::= int_param_decl_list ;
int_param_decl_list -> INT_PARAM_DECL * ... ;
INT_PARAM_DECL ::= int_param_decl;
int_param_decl -> MODE TYPE_SPEC ;
MODE ::= in_mode out_mode inout_mode ;
FUNCTION_DEF ::= forward_function_def function_def ;
forward_function_def -> NAME TYPE_SPEC_LIST TYPE_SPEC_LIST ;
-- YSC 3/30/95 add SPECIAL_VARIABLE_LIST.
function_def ->
 FUNCTION_HEADER SPECIAL_VARIABLE_LIST ORDINARY_VARIABLE_LIST EXPRESSION_LIST ;
-- SYSC 6/28/95 if there are both SPECIAL_VARIABLE_LIST and ORDINARY_VARIABLE_
-- SPECIAL_VARIABLE_LIST should be in the front.
FUNCTION_HEADER ::= function_header ;
function_header -> NAME PARAM_DECL_LIST TYPE_SPEC_LIST ;
PARAM_DECL_LIST ::= param_decl_list;
param_decl_list -> PARAM_DECL * ... ;
PARAM_DECL ::= param_decl ;
param_decl -> NAME_LIST TYPE_SPEC ;
-- YSC 3/30/95 add the following line.
       SPECIAL_VARIABLE_LIST ::= special_variable_list;
       special_variable_list -> NAME * ...;
```

```
      ORDINARY_VARIABLE_LIST ::= ordinary_variable_list;
      ordinary_variable_list -> NAME * ...;
--    SPECIAL_VARIABLE ::= special_variable;
--    special_variable -> NAME;

 end chapter ;

chapter 'expressions'
  rules
-- YSC
      <Opt_Ordinary_variable_List> ::=  ;
              ordinary_variable_list-list(())

      <Opt_Ordinary_variable_List> ::=
 "ordinary" <Ordinary_variable_List> "end"
      "ordinary" ";" ;
<Ordinary_variable_List>

-- YSC 6-28-95 terminal of Ordinary_variable_List
<Ordinary_variable_List> ::= <Ordinary_variable>;
           ordinary_variable_list-list((<Ordinary_variable>))

<Ordinary_variable_List> ::=
<Ordinary_variable_List> "," <Ordinary_variable>;
   ordinary_variable_list-post(<Ordinary_variable_List>,
<Ordinary_variable>)

       <Ordinary_variable> ::= <Name>;
              <Name>


-- YSC 3/30/95
      <Opt_Special_variable_List> ::=  ;
              special_variable_list-list(())

      <Opt_Special_variable_List> ::= "special" <Special_variable_List> "end"
      "special" ";" ;
<Special_variable_List>

-- YSC 6-28-95 terminal of Special_variable_List
<Special_variable_List> ::= <Special_variable>;
           special_variable_list-list((<Special_variable>))

<Special_variable_List> ::=
 <Special_variable_List> "," <Special_variable>;
   special_variable_list-post(<Special_variable_List>,<Special_variable>)

       <Special_variable> ::= <Name>;
              <Name>


      <Expression_List> ::= <Expression> ;
expression_list-list((<Expression>))

<Expression_List> ::= <Expression_List> "," <Expression> ;
expression_list-post(<Expression_List> , <Expression>)


<Expression> ::= <Expression> "|" <Logical_Relation> ;
binary(<Expression>, or_op(), <Logical_Relation>)

<Expression> ::= <Expression> "&" <Logical_Relation> ;
binary(<Expression>, and_op(), <Logical_Relation>)

<Expression> ::= <Logical_Relation> ;
<Logical_Relation>

<Logical_Relation> ::= <Logical_Relation> "<" <Add_Expression> ;
```

```
binary(<Logical_Relation>, lt_op(), <Add_Expression>)

<Logical_Relation> ::= <Logical_Relation> "<=" <Add_Expression> ;
binary(<Logical_Relation>, lte_op(), <Add_Expression>)

<Logical_Relation> ::= <Logical_Relation> ">" <Add_Expression> ;
binary(<Logical_Relation>, gt_op(), <Add_Expression>)

<Logical_Relation> ::= <Logical_Relation> ">=" <Add_Expression> ;
binary(<Logical_Relation>, gte_op(), <Add_Expression>)

<Logical_Relation> ::= <Logical_Relation> "=" <Add_Expression> ;
binary(<Logical_Relation>, eq_op(), <Add_Expression>)

<Logical_Relation> ::= <Logical_Relation> "~=" <Add_Expression> ;
binary(<Logical_Relation>, neq_op(), <Add_Expression>)

<Logical_Relation> ::= <Add_Expression> ;
<Add_Expression>

<Add_Expression> ::= <Add_Expression> "||" <Add_Exp> ;
binary(<Add_Expression>, concat_op(), <Add_Exp>)

<Add_Expression> ::= <Add_Exp> ;
<Add_Exp>

<Add_Exp> ::= <Add_Exp> "+" <Mult_Expression> ;
binary(<Add_Exp>, plus_op(), <Mult_Expression>)

<Add_Exp> ::= <Add_Exp> "-" <Mult_Expression> ;
binary(<Add_Exp>, minus_op(), <Mult_Expression>)

<Add_Exp> ::= <Mult_Expression> ;
<Mult_Expression>

<Mult_Expression> ::= <Mult_Expression> "*" <Power> ;
binary(<Mult_Expression>, mult_op(), <Power>)

<Mult_Expression> ::= <Mult_Expression> "/" <Power> ;
binary(<Mult_Expression>, div_op(), <Power>)

<Mult_Expression> ::= <Power> ;
 <Power>

<Power> ::= <Power> "**" <Unary> ;
binary(<Power>, exp_op(), <Unary>)

<Power> ::= <Unary> ;
<Unary>

<Unary> ::= "~" <Unary2> ;
unary(not_op(), <Unary2>)

<Unary> ::= "+" <Unary2> ;
unary(plus_op(), <Unary2>)

<Unary> ::= "-" <Unary2> ;
unary(minus_op(), <Unary2>)

<Unary> ::= <Unary2>;
<Unary2>

<Unary2> ::= <TermC> ;
<TermC>
        ......

<Array_Ref> ::= <Term> "[" <Selector> "]" ;
array_ref(<Term>, <Selector>)
```

```
-- <Array_Ref> ::= <Term> "[" <Selector_Part_List> <Diag_Spec_List> "]" ;
-- array_ref(<Term>,
-- selector(<Selector_Part_List>,<Diag_Spec_List>))

-- <Array_Ref> ::= <Term> "["<Name> "in" <Triplet>"]";
-- array_ref(<Term>,
-- selector(
--  selector_part_list-list((triplet2(<Name>,<Triplet>))),
--  diag_spec_list-list(()))

-- <Array_Ref> ::= <Term> "["<Name> "in" <Triplet><Opt_Diag_Spec_List>"]";
-- array_ref(<Term>,
-- selector(
--  selector_part_list-list((triplet2(<Name>,<Triplet>))),
--  <Opt_Diag_Spec_List>))

-- <Array_Ref> ::= <Term> "[" <Selector_Part_List>"," <Selector_Part>
-- "]";
-- array_ref(<Term>,
-- selector(selector_part_list-post(<Selector_Part_List>,
--  <Selector_Part>),
-- diag_spec_list-list(()))

<Array_Gen> ::= "array" <Type_Spec> "["  "]" ;
array_gen(<Type_Spec>, size_descr_list-list(()),
                        array_part_list-list(()))

      <Array_Gen> ::= "array" <Type_Spec> "[" <Array_Part_List>
      <Opt_Semicolon> "]";
              array_gen(<Type_Spec>, size_descr_list-list(()),
      <Array_Part_List>)

      <Array_Gen> ::= "array" <Type_Spec> "[" <Size_Descriptor> ":"
      <Array_Part_List> <Opt_Semicolon> "]";
              array_gen(<Type_Spec>,
  <Size_Descriptor>, -- change from YSC
  <Array_Part_List>)

<Array_Update> ::= <Term> "[" <Update_Part_List> <Opt_Semicolon> "]" ;
array_update(<Term>, <Update_Part_List>)

<Selector> ::= <Selector_Part_List> <Opt_Diag_Spec_List> ;
selector(<Selector_Part_List>, <Opt_Diag_Spec_List>)

<Selector_Part_List> ::= <Selector_Part> ;
selector_part_list-list((<Selector_Part>))

<Selector_Part_List> ::= <Selector_Part_List> "," <Selector_Part> ;
selector_part_list-post(<Selector_Part_List>, <Selector_Part>)

<Selector_Part> ::= <Expression> ;
<Expression>

<Selector_Part> ::= <Selector_Part1> ;
<Selector_Part1>

<Selector_Part1> ::= <Triplet>;
    triplet2(no_name(), <Triplet>)

<Selector_Part1> ::= <Selector_Part2> ;
<Selector_Part2>

<Selector_Part2> ::= <Name> "in" <Triplet> ;
triplet2(<Name>, <Triplet>)

<Opt_Diag_Spec_List> ::= ;
```

```
diag_spec_list-list(())

<Opt_Diag_Spec_List> ::= "{" <Diag_Spec_List> "}" ;
<Diag_Spec_List>

<Diag_Spec_List> ::= <Diag_Spec> ;
diag_spec_list-list((<Diag_Spec>))

<Diag_Spec_List> ::= <Diag_Spec_List> "," <Diag_Spec> ;
diag_spec_list-post(<Diag_Spec_List>, <Diag_Spec>)

<Diag_Spec> ::= <Name> "dot" <Dot_Value_List> ;
                diag_spec(<Name>, <Dot_Value_List>)

        <Dot_Value_List> ::= <Name>;
                name_list-list((<Name>))

        <Dot_Value_List> ::= <Name> "dot" <Dot_Value_List>;
                name_list-pre(<Name>,<Dot_Value_List>)


<Size_Descriptor> ::= <Selector_Part1_List> ;
<Selector_Part1_List>

<Selector_Part1_List> ::= <Selector_Part1> ;
size_descr_list-list((<Selector_Part1>))

<Selector_Part1_List> ::= <Selector_Part1_List> "," <Selector_Part1> ;
size_descr_list-post(<Selector_Part1_List>, <Selector_Part1>)


<Array_Part_List> ::= <Array_Part> ;
array_part_list-list((<Array_Part>))

<Array_Part_List> ::= <Array_Part_List> ";" <Array_Part> ;
array_part_list-post(<Array_Part_List>, <Array_Part>)

<Array_Part> ::= <Expression_List> ;
array_part(no_placement(), <Expression_List>)

<Array_Part> ::= <Placement> <Expression_List> ;
array_part(<Placement>, <Expression_List>)

<Placement> ::= "[" <Selector> "]" ;
<Selector>

<Placement> ::= "[" "otherwise" "]" ;
otherwise()

<Update_Part_List> ::= <Update_Part> ;
update_part_list-list((<Update_Part>))

<Update_Part_List> ::= <Update_Part_List> ";" <Update_Part> ;
update_part_list-post(<Update_Part_List>, <Update_Part>)

<Update_Part> ::= <Selector> ":" <Expression_List> ;
update_part( <Selector>, <Expression_List>)

-- <Complex_Ref>  ::= <Term> "." "imag" ;
-- complex_ref(<Term>, imag())

-- <Complex_Ref>  ::= <Term> "." "real" ;
-- complex_ref(<Term>, real())


<Complex_Gen> ::= "(" <Expression> "," <Expression> ")" ;
complex_gen(<Expression>.1, <Expression>.2)
```

```
<Conditional_Expression> ::= "if" <Expression> "then" <Expression_List>
<Opt_Elseif_List> "else" <Expression_List>
"end" "if" ;
conditional(<Expression>, <Expression_List>.1,
<Opt_Elseif_List>,<Expression_List>.2)

<Opt_Elseif_List> ::= ;
elseif_list-list(())

<Opt_Elseif_List> ::= <Elseif_List> ;
<Elseif_List>

<Elseif_List> ::= <Elseif> ;
elseif_list-list((<Elseif>))

<Elseif_List> ::= <Elseif_List> <Elseif> ;
elseif_list-post(<Elseif_List>, <Elseif>)

<Elseif> ::= "elseif" <Expression> "then" <Expression_List> ;
elseif(<Expression>, <Expression_List>)

<Let_Expression> ::= "let" <Decl_Def_Part> "in" <Expression_List>
"end" "let" ;
let(<Decl_Def_Part>, <Expression_List>)

<Decl_Def_Part> ::= <Decl_Def_List> <Opt_Semicolon> ;
 <Decl_Def_List>

 <Decl_Def_List> ::= <Decl_Def> ;
decl_def_list-list((<Decl_Def>))

 <Decl_Def_List> ::= <Decl_Def_List> ";" <Decl_Def> ;
decl_def_list-post(<Decl_Def_List>, <Decl_Def>)

<Decl_Def> ::= <Decl_List> ":=" <Expression_List> ;
decl_def(<Decl_List>, <Expression_List>)

<Decl_List> ::= <Decl> ;
decl_list-list((<Decl>))

-- conflit entre "," ici et "," dans <Decl_Id_List>
<Decl_List> ::= <Decl_List> "," <Decl> ;
decl_list-post(<Decl_List>, <Decl>)

<Decl> ::= <Decl_Id_List> ;
decl(<Decl_Id_List>, no_type())

<Decl> ::= <Decl_Id_List> ":" <Type_Spec> ;
decl(<Decl_Id_List>, <Type_Spec>)

<Decl_Id_List> ::= <Decl_Id> ;
decl_id_list-list((<Decl_Id>))

<Decl_Id_List> ::= <Decl_Id_List> "," <Decl_Id> ;
decl_id_list-post(<Decl_Id_List>, <Decl_Id>)

<Decl_Id> ::= <Name> ;
<Name>

<Decl_Id> ::= <Function_Header> ;
<Function_Header>

<Case_Expression> ::=  "case" <Expression> "of" <Case_Part_List>
<Opt_Semicolon> "end" "case" ;
case(<Expression>, <Case_Part_List>,expression_list-list(()))

<Case_Expression> ::=  "case" <Expression> "of" <Case_Part_List>
";" "otherwise" ":" <Expression_List> <Opt_Semicolon>
```

```
"end" "case" ;
case(<Expression>, <Case_Part_List>, <Expression_List>)

<Case_Part_List> ::= <Case_Part> ;
case_part_list-list((<Case_Part>))

<Case_Part_List> ::= <Case_Part_List> ";" <Case_Part> ;
case_part_list-post(<Case_Part_List>, <Case_Part>)

<Case_Part> ::= <Pattern_List> ":" <Expression_List> ;
case_part(<Pattern_List>, <Expression_List>)

<Pattern_List> ::= <Pattern> ;
pattern_list-list((<Pattern>))

<Pattern_List> ::= <Pattern_List> "," <Pattern> ;
pattern_list-post(<Pattern_List>, <Pattern>)

<Pattern> ::= "false" ;
false()

<Pattern> ::= "true" ;
true()

<Pattern> ::= <Int_Const> ;
<Int_Const>

<Pattern> ::= <Character_Const> ;
<Character_Const>

<Pattern> ::= <Character_String_Const> ;
<Character_String_Const>


<Pattern> ::= <Type_Spec> ;
<Type_Spec>

<For_Expression> ::= <Opt_For_Top> <For_Body> <For_Bottom> ;
for(<Opt_For_Top> , <For_Body>, <For_Bottom>)

<Opt_For_Top> ::= ;
no_for_top()

<Opt_For_Top> ::= "for" <In_Exp_List> <Opt_Decl_Def_Part>
  <Opt_For_Test> ;
for_top(<In_Exp_List>, <Opt_Decl_Def_Part>,
  <Opt_For_Test>)

<Opt_For_Top> ::= "for" <Decl_Def_Part> <Opt_For_Test> ;
for_top(dot_in_list-list(()), <Decl_Def_Part>,
<Opt_For_Test>)

<Opt_For_Top> ::= <For_Test> ;
for_top(dot_in_list-list(()), decl_def_list-list(()),
<For_Test>)

<In_Exp_List> ::= <In_Exp> ;
dot_in_list-list((<In_Exp>))


<In_Exp_List> ::= <In_Exp> <Dot_List> ;
        dot_in_list-pre(<In_Exp>, <Dot_List>)


<In_Exp_List> ::= <In_Exp> <Cross_List> ;
        cross_in_list-pre(<In_Exp>, <Cross_List>)
```

```
<Dot_List> ::= "dot" <In_Exp>;
dot_in_list-list((<In_Exp>))

<Dot_List> ::= <Dot_List> "dot" <In_Exp>  ;
dot_in_list-post(<Dot_List>,<In_Exp>)

<Cross_List> ::= "cross" <In_Exp> ;
cross_in_list-list((<In_Exp>))

<Cross_List> ::= <Cross_List> "cross" <In_Exp>  ;
cross_in_list-post(<Cross_List>, <In_Exp>)

.........

end chapter ;

chapter 'values'
abstract syntax
VALUES ::= values;
values -> VALUE * ... ;
VALUE ::= CONSTANT array union record stream closure ;

BOOLEAN ::= true false;
array -> DIM DIM_CONTENT ;
DIM ::= dim ;
dim -> implemented as INTEGER ;
DIM_CONTENT ::= dim_content ;
dim_content -> LOWER UPPER ELTS;
LOWER ::= lower;
UPPER ::= upper ;
lower ->  implemented as INTEGER ;
upper ->  implemented as INTEGER ;

-- YSC 12-20
ELTS ::= elts;
elts -> ELT * ...;

ELT ::= DIM_CONTENT VALUE;

union -> TAG_ID VALUE;

record -> FIELD + ... ;
FIELD ::= field ;
field -> FIELD_ID VALUE;

stream -> STREAM_ELT * ... ;
STREAM_ELT ::= VALUE TRIPLET ;
closure -> LAMBDA ENV ENV FUN_PATH ;

FUN_PATH ::= fun_path;
fun_path -> implemented as TREE;
LAMBDA ::= assign_function_def ;
ENV ::= env;
        env -> PAIR * ...;
PAIR ::= pair;
pair -> NAME VALUE;
assign_function_def -> FUNCTION_HEADER EXPRESSION_LIST ;
end chapter ;

chapter 'entry points'
rules
         <axiom> ::= "[COMPILATION_UNIT]" <Compilation_Unit>;
 <Compilation_Unit>

  <axiom> ::= "[DECLARATION]" <Int_Declaration>;
 <Int_Declaration>

 <axiom> ::= "[TYPE_DEF]" <Type_Def>;
```

```
<Type_Def>

<axiom> ::= "[TYPE_SPEC]" <Type_Spec>;
<Type_Spec>

<axiom> ::= "[FIELD_SPEC_LIST]" <Field_Spec_List>;
<Field_Spec_List>

<axiom> ::= "[DIM_SPEC_LIST]" <Dim_Spec>;
<Dim_Spec>

        <axiom> ::= "[FIELD_SPEC]" <Field_Spec>;
<Field_Spec>

<axiom> ::= "[TAG_SPEC]" <Tag_Spec>;
<Tag_Spec>

<axiom> ::= "[TRIPLET]" <Triplet> ;
<Triplet>

      <axiom> ::= "[EXPRESSION_LIST]" <Expression_List> ;
                  <Expression_List>

      <axiom> ::= "[EXPRESSION]" <Expression> ;
                  <Expression>

      <axiom> ::= "[OPT_EXPRESSION]" <Opt_Expression> ;
                  <Opt_Expression>

<axiom> ::= "[DECL_ID]" <Decl_Id> ;
<Decl_Id>

<axiom> ::= "[NAME]" <Name> ;
<Name>
end chapter ;
end definition
```

Reference List

[1] Y. Ben-Asher, G. Runger, A. Schuster, and R. Wilhelm, "2 dt-fp: A parallel functional programming language on two dimensional data," *International Journal of Parallel Programming*, vol. 23, no. 5, pp. 389–422, Oct. 1995.

[2] J.-L. Gaudiot and C. Kim, "Data-driven and multithreaded architectures for high-performance computing," in *Parallel Computers: Theory and Practice* (T. Casavant and P. Tvrdik, eds.), ch. 4, IEEE Computer Society Press: Washington, DC, 1993.

[3] C. Kim, "Functional programming and fine-grain multithreading for high-performance parallel computing," Tech. Rep. PhD. Disseration, University of Southern California EE-system, Aug. 1994.

[4] J. L. Gaudiot and Y. Wei, "Token relabeling in a tagged-token data-flow architecture," *IEEE Trans. Computers*, vol. 38, no. 9, Sep. 1989.

[5] C. Kim and J. Gaudiot, "A scheme to extract run-time parallelism from sequential loops," in *Proc. 5th ACM Int'l Conf. Supercomputing*, Jun. 1991.

[6] C. Kim, J.-L. Gaudiot, and W. Proskurowski, "Programmability and performance issues: the case of an iterative partial differential equation solver," in *Proc. Sisal '93*, San Diego, CA, Oct. 1993.

[7] P. Trinder, "Referentially transparent database," in *Proc. 1989 Glasgow Workshop: Functional Programming*, pp. 142–154, Fraserburgh, Scotland, Aug. 1989.

[8] K. M. Chandy and I. Foster, "A notation for deterministic cooperating processes," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 8, pp. 863–871, Aug. 1995.

[9] K. M. Chandy and J. Misra, *Parallel Program Design.* Addison-Wesley Publishing Company, Inc.: Reading, MA, 1988.

[10] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing.* McGraw-Hill Publishing Company, Inc.: New York, NY, 1984.

[11] D. Abramson and A. McKay, "Evaluating the performance of a Sisal implementation of the abingdon cross image processing benchmark," *International Journal of Parallel Programming*, vol. 23, no. 2, pp. 105–134, Aug. 1995.

[12] P. Henderson, *Functional Programming application and implementation.* Prentice-Hall, Inc.: Englewood Cliffs, NJ, 1980.

[13] P. Hudak in *Para-Functional Programming in Haskell* (B. Szymanski, ed.), ACM Press: New York, N.Y., 1991.

[14] E. L. Lafferty, M. J. Prelle, M. C. Michaud, and J. B. Goethert, "Parallel computing an introduction," tech. rep., Noyes Data Corporation, New Jersey, 1993.

[15] G. Almasi and A. Gottlieb, *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company: Menlo Park, CA, 1994.

[16] J. L. Gaudiot, "Structure handling in data-flow systems," *IEEE Trans. Computers*, vol. 35, no. 6, pp. 489–502, Jun. 1986.

[17] K. Hwang and D. Degroot, *Parallel Processing for Supercomputers and Artificial Intelligence.* McGraw-Hill Publishing Company, Inc.: New York, NY, 1989.

[18] A. Wikstrom, *Functional Programming Using Standard ML.* Prentice-Hall, Inc.: Englewood Cliffs, NJ, 1992.

[19] J. Backus, "Function-level computing," *IEEE Spectrum Magazine*, vol. 19, no. 8, no. 8, pp. 22–27, 1982.

[20] D. Engelhardt and A. Wendelborn, "Investigating the memory performance of the optimising Sisal compiler," in *Proc. the Second Sisal Users' Conference* (J. Feo, C. Frerking, and P. Miller, eds.), pp. 257–270, Dec. 1992.

[21] A. Wendelborn and H. Garsden, "Exploring the stream data type in Sisal and other languages," in *IFIP Transactions: Architectures and Compilation Techniques for Fine and Medium Grain Parallelism* (M. Cosnard, K. Ebcioğlu, and J.-L. Gaudiot, eds.), pp. 283–294, North-Holland: IFIP, Jan. 1993.

[22] P. Hudak, "Functional programming languages," *ACM Computing Surveys*, vol. 21, no. 3, pp. 359–411, Sep. 1989.

[23] L. McLoughlin and E. Hayes, "Imperative effects from a pure functional language," in *Proc. 1989 Glasgow Workshop: Functional Programming*, pp. 157–169, Fraserburgh, Scotland, Aug. 1989.

[24] S. P. Jones and P. Wadler, "Imperative functional programming," in *ACM Principles of Programming Languages*, 1993.

[25] J. Launchbury, "Lazy imperative programming," tech. rep., Department of Computer Science, University of Glasgow, Dec. 1993.

[26] P. Wadler, "The essence of functional programming," in *ACM Principles of Programming Languages*, 1992.

[27] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the Sisal language project," Tech. Rep. Technical Report, UCRL-102440, Lawrence Livermore National Laboratory, 1990.

[28] R. R. Oldehoeft and J. R. McGraw, "Mixed applicative and imperative programs," *Parallel Computing*, vol. 13, no. 2, no. 2, pp. 175–191, 1990.

[29] A.P.W. Bohm R.R. Oldehoeft and D.C. Cann and J.T. Feo, Computer Science Department, Colorado State University, and Computing Research Group, Lawrence Livermore National Laboratory, *Sisal Reference Manual Language Version 2.0*, 1990.

[30] Arvind and R. Nikhil, "I-structures: data structures for parallel computing," *ACM Trans. Programming Languages and Systems*, vol. 11, no. 4, pp. 598–632, Oct. 1989.

[31] M. P. Jones and L. Duponcheel, "Composing monads," Tech. Rep. Research Report YALEU/DCS/RR-1004, Yale University, Dec. 1993.

[32] P. Wadler, "Comprehending monads," in *Proc. ACM Conf. Lisp and Functional Programming*, Nice, 1990.

[33] B. Meyer, *Introduction to the Theory of Programming Languages.* Prentice-Hall, Inc.: Englewood Cliffs, NJ, 1991.

[34] A. Berstein, "Analysis of programs for parallel processing," *IEEE Trans. Computers*, pp. 746–757, 1966.

[35] K. Hwang, *Advanced Computer Architecture with Parallel Programming.* McGraw-Hill Publishing Company, Inc.: New York, NY, 1993.

[36] Y. S. Chen and J. L. Gaudiot, "Parallelism detection algorithm for extended Sisal programs in centaur.," in *Proc. 8th Int'l Conf. Parallel and Distributed Computing Systems*, pp. 628–633, Sep. 1995.

[37] T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, "Centaur: the system," Tech. Rep. Technical Report, INRIA, Sophia-Antipolis and INRIA, Rocquencourt, 1992.

[38] J. L. Gaudiot and Y. S. Chen, "Specification of the array semantics for Sisal 2.0," Tech. Rep. Technical Report 94-28, University of Southern California EE-system, July 1994.

[39] I. Attali, D. Caromel, Y. S. Chen, J. L. Gaudiot, and A. Wendelborn, "A formal semantics for Sisal arrays," in *Proc. Joint Conf. Information Sciences*, Sep. 1995.

[40] J. T. Feo, "The livermore loops in Sisal," tech. rep., Lawrence Livermore National Laboratory, 1987.

[41] I. Attali, D. Caromel, Y. Chen, J. Gaudiot, and A. Wendelborn, "A formal semantics for Sisal arrays," Tech. Rep. Technical Report, University of Southern California EE-system, Dec 1994.

[42] E. Horowitz and S. Sahni, *Fundamentals of data structures in Pascal.* Computer Science Press: New York, NY, 1989.

[43] E. Horowitz, S. Anderson-Freed, and S. Sahni, *Fundamentals of data structures in C.* Computer Science Press: New York, NY, 1993.

[44] E. Rich and K. Knight, *Artificial Integellence.* McGraw-Hill Publishing Company, Inc.: New York, NY, 1991.