Escape constructs in data-parallel languages: semantics and proof system

Luc Bougé, Gil Utard

Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Escape constructs in data-parallel languages: semantics and proof system

Luc Bougé
Gil Utard

June 1994

Research Report N$^{\mathrm{o}}$ 94-18

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00   Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Escape constructs in data-parallel languages: semantics and proof system

Luc Bougé
Gil Utard

June 1994

Abstract

We describe a simple data-parallel kernel language which encapsulates the main data-parallel control structures found in high-level languages such as MasPar's MPL or the recent HyperC. In particular, it includes the concept of *data-parallel escape*, which extends the break and continue constructs of the scalar C language. We give this language a *natural semantics*, we define a notion of *assertion* and describe an assertional proof system. We demonstrate its use by proving a small data-parallel Mandelbrot-like program.

Citation: This work has been submitted for presentation at the *14th Conference on the Foundations of Software Technology and Theoretical Computer Science*, December 15–17, 1994, Madras, India.

Keywords: Concurrent Programming; Specifying and Verifying and Reasoning about Programs; Semantics of Programming Languages; Data-Parallel Languages; Proof System; Hoare Logic.

Résumé

Nous décrivons un langage minimal qui capture la sémantique des structures de contrôle des langages data-parallèles tels que MPL de MasPar ou HyperC. En particulier, il étend le concept d'*échappement* du langage *C* scalaire, tel que le break ou continue, au cas data-parallèle. Nous en définissons une *sémantique naturelle*, puis nous définissons une notion d'*assertion* et décrivons un système de preuve de programmes par assertions selon la méthode axiomatique de Hoare. La mise en œuvre du système est illustrée par un exemple.

Référence à citer. Ce travail a été soumis pour une présentation à la *14th Conference on the Foundations of Software Technology and Theoretical Computer Science*, December 15–17, 1994, Madras, India

Mots-clés: programmation parallèle ; spécification et validation de programmes ; sémantique des langages de programmation ; langages data-parallèles ; système de preuve ; logique de Hoare.

Escape constructs in data-parallel languages: semantics and proof system

Luc Bougé*‡, Gil Utard*

June 1, 1994

Abstract

We describe a simple data-parallel kernel language which encapsulates the main data-parallel control structures found in high-level languages such as MasPar's MPL or the recent HyperC. In particular, it includes the concept of *data-parallel escape*, which extends the `break` and `continue` constructs of the scalar C language. We give this language a *natural semantics*, we define a notion of *assertion* and describe an assertional proof system. We demonstrate its use by proving a small data-parallel Mandelbrot-like program.

Keywords: Concurrent Programming; Specifying and Verifying and Reasoning about Programs; Semantics of Programming Languages; Data-Parallel Languages; Proof System; Hoare Logic.

Citation: This work has been submitted for presentation at the *14th Conference on the Foundations of Software Technology and Theoretical Computer Science*, December 15–17, 1994, Madras, India.

*LIP, ENS Lyon, 46 Allée d'Italie, F-69364 Lyon Cédex 07, France.

‡Authors contact: Luc Bougé (`Luc.Bouge@lip.ens-lyon.fr`). This work has been partly supported by the French CNRS Coordinated Research Program on Concurrency, Communication and Cooperation C^3, and Department of Defense DRET contract 91/1180.

Contents

1 Introduction

The impressive effort put in the design and the implementation of the High Performance Fortran (HPF) language [6] in the past year has brought data-parallelism to the forefront of the research scene. Data-parallel programming appears today as a major advance in the long quest towards a portable parallel programming environment, available from low cost workstation clusters to massively parallel computers. We are now witnessing the emergence of a number of data-parallel languages, most of them derived from Fortran or C, and mainly designed by the constructors of massively parallel machines for their customers. Unfortunately, their design has often been primarily motivated by pragmatic and short-term considerations, and comparatively few studies have been done on their principles, on their semantic expressivity, or on the associated program validation methods. This results in poor language design, semantic "hand-waving" and pitfalls, unmaintainable programs and, ultimately, a waste of time and money.

Current data-parallel languages can be classified into two categories, depending on the diversity of their data-parallel control structures.

- ▹ *Low-level languages*, such as HPF or Thinking Machine's C*, offer parallel data-types (distributed arrays, shapes, etc.) and assignment commands between parallel objects (possibly including rearrangement). But the overall control structure is still scalar, inherited from the original language. There is no specific data-parallel control besides the conditioning **where** construct which restricts the current extent of parallelism. Such languages are very close to the 20-year old Actus language of Perrot [13], probably the first attempt towards a semantic approach of data-parallelism.
- ▹ *High-level languages*, such as MasPar's MPL or the recent HyperC [12], define, besides the scalar control structure inherited from the original language, a rich set of data-parallel control structures. They include the data-parallel extensions of usual scalar constructs such as **for**, **while**, **switch**, and their associated *non-local control-transfer commands* **break** and **continue**. This additional semantic facility has proved to be appropriate for a clean description of many data-parallel algorithms, in the same way as using the C **break** construct often leads to a clearer coding. In this respect, these languages can be called fully *data-parallel*, as they go much beyond embedding data-parallel assignments into a scalar control harness.

Several researchers have attempted to give a formal semantics to such data-parallel languages, and to use it to prove programs correct. Most works have focused on the data-parallel assignment (see for instance [14], and to some extent [7]), that is a low-level approach in the above classification. An early attempt for the Actus language can be found in [5]: there, the entire Actus language is considered, at the price of considerable technical complexity. In contrast, we have shown in our previous papers that it is possible to define a *kernel data-parallel language*, called $\mathcal{L}$, which encapsulates the main features of the low-level languages [4]. We have described a natural semantics for it, and an assertional proof system [3]. Moreover, we have shown the possibility of defining a weakest precondition (WP) calculus for $\mathcal{L}$ [2]. It can be used as a basis to demonstrate the (relative) completeness of this proof system. It also opens the way to a computer-aided validation tool for $\mathcal{L}$ programs based on the automatic generation of verification conditions [8, 9].

This contribution of this paper is to extend these results to the case of *high-level languages*. First, we describe an extended version of the $\mathcal{L}$ kernel language, called $\mathcal{L}'$, which captures the notion of *data-parallel non-local control transfer*. We give it a natural semantics based on the multi-context approach of [4]. We define an extended notion of assertion, and we describe the associated proof rules. This is illustrated by the proof of a simple data-parallel factorial program.

2 The $\mathcal{L}$ language

An extensive presentation of the $\mathcal{L}$ language can be found in [4]. For the sake of completeness, we briefly recall its natural semantics as described in [3].

2.1 Informal description

In the data-parallel programming model, the basic objects are arrays with parallel access. Two kinds of actions can be applied to these objects: *componentwise* operations, and global *rearrangements*. A program is a sequential composition of such actions. Each action is associated with the set of array indices at which it is applied. An index at which an action is applied is said to be *active*. Other indices are said to be *idle*. The set of active indices is called the *activity context* or the *extent of parallelism*. It can be seen as a boolean array where *true* denotes activity and *false* idleness.

The $\mathcal{L}$ language is designed as a common kernel of data-parallel languages like C^* [15], HyperC [12] or MPL [10]. We do not consider the scalar part of these languages, mainly imported from the C language. For the sake of simplicity, we consider only arrays of dimension one, also called *vectors*, indexed by integers. It follows that all variables of $\mathcal{L}$ are parallel, and all the objects are vectors of scalars, with one component at each index. As a convention, the parallel objects are denoted with uppercase letters. The legal expressions are *pure* expressions, i.e. without side effects, defined like the *pure* functions of HPF. The value of a pure expression at index u only depends on the values of the variables at index u. The expressions are evaluated by applying operators *componentwise* to parallel values. We do not further specify the syntax and semantics of such expressions. The component of parallel object X located at index u is denoted by $X|_u$. We introduce a special vector constant called *This*. The value of its component at each index u is the value u itself: $\mathit{This}|_u = u$. Note that *This* is a pure expression and that all constructs defined here are *deterministic*.

Assignment: $X := E$. At each active index u, component $X|_u$ is updated with the local value of *pure* expression E.

Communication: get X from A into Y. At each active index u, *pure* expression A is evaluated to an index v, then component $Y|_u$ is updated with the value of component $X|_v$. We always assume that v is a valid index. (Notice that the remote expression X is restricted to be a variable.)

Sequencing: $S; T$. The execution of the actions of T starts on the termination of the last action of S.

Iteration: loop B do S end. The actions of S are repeatedly executed with the current extent of parallelism, until *pure* boolean expression B evaluates to false at each currently active index.

Conditioning: where B do S end. The body S of a conditioning block is executed in a new activity context defined as follows. The initially idle indices remain idle during the execution S. The initially active indices where *pure* boolean expression B evaluates to false are turned idle

```
M := N; R := N;
loop (M > 1) do
    where (M > 1) do
        M := M − 1;
        R := R × M
    end
end
```

Figure 1: The data-parallel factorial, $\mathcal{L}$ version.

during the execution of S: they are called *deselected*. The remaining ones remain active during the execution of S: they are called *selected*. The initial activity context is restored on termination of S.

Note that the MPL data-parallel while construct can be expressed by a where nested in a loop. As *This* is a pure $\mathcal{L}$ expression, note also that the local computations may depend on the value of the local index.

In this paper, we will consider the running example shown in Figure 1. Given an initial integer vector N with all components $N|_u \geq 1$, we want to compute an integer vector R such that, at each index u, $R|_u = \text{factorial}(N|_u)$. The idea is to build the decreasing product chain at each index until 1 is reached, and then to remain in active wait until 1 is reached at *all* indices.

2.2 A natural semantics for $\mathcal{L}$

We describe the semantics of $\mathcal{L}$ in the style of Kahn and Plotkin's *natural semantics* by induction on the syntax of $\mathcal{L}$.

An *environment* σ is a function from identifiers to vector values. The set of environments is denoted by Env. For convenience, we extend the environment functions to parallel expressions: $\sigma(E)$ denotes the value obtained by evaluating parallel expression E in environment σ. We do not specify any further the internals of expressions. Note that $\sigma(\mathit{This})|_u = u$ by definition.

Definition 1 (Pure expression) *A parallel expression E is* pure *if for any index u, and any environments σ and σ',*

$$(\forall X : \sigma(X)|_u = \sigma'(X)|_u) \Rightarrow (\sigma(E)|_u = \sigma'(E)|_u).$$

Let σ be an environment, X a vector variable and V a vector value. We denote by $\sigma[X \leftarrow V]$ the new environment σ' where $\sigma'(X) = V$ and $\sigma'(Y) = \sigma(Y)$ for all $Y \neq X$.

A *context* c is a boolean vector. It specifies the activity at each index: $c|_u$ is $true$ iff index u is active. The set of contexts is denoted by Ctx. We distinguish a particular context denoted by $True$ whose components all have value $true$. For the sake of consistency with Section 3.2 we introduce the notation *active*:

$$active \qquad \equiv \qquad c$$

A *state* is a pair made of an environment and a context. The set of states is denoted by $State$: $State = (Env \times Ctx) \cup \{\perp\}$ where $\perp$ denotes the undefined state.

The semantics $[\![S]\!]$ of a program S is a *strict* function from $State$ to $State$. $[\![S]\!](\perp) = \perp$, and $[\![S]\!]$ is extended to sets of states as usual. The following paragraphs define $[\![\ ]\!]$ for $\mathcal{L}$ programs.

Assignment. At each active index, the component of the parallel variable is updated with the new value.

$$[\![X := E]\!](\sigma, c) = (\sigma', c)$$

with $\sigma' = \sigma[X \leftarrow V]$ where $V|_u = \sigma(E)|_u$ if $active|_u$, and $V|_u = \sigma(X)|_u$ otherwise. The activity context is preserved. Notice that since E is pure, the evaluation of $\sigma(E)|_u$ requires no communications, it is local.

Communication. It acts very much as an assignment, except that the assigned value is the value of another component.

$$[\![\textsf{get } X \textsf{ from } A \textsf{ into } Y]\!](\sigma, c) = (\sigma', c),$$

with $\sigma' = \sigma[Y \leftarrow V]$ where $V|_u = \sigma(X)|_{\sigma(A)|_u}$ if $active|_u$, and $V|_u = \sigma(Y)|_u$ otherwise. Again, the evaluation of $\sigma(A)|_u$ is local, and context is preserved.

Sequencing. Sequential composition denotes functional composition.

$$[\![S; T]\!](\sigma, c) = [\![T]\!]([\![S]\!](\sigma, c)).$$

Iteration: Iteration is expressed by classical loop unfolding. It terminates when the *pure* boolean expression B evaluates to false at each active index.

$$[\![\textsf{loop } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = \begin{cases} [\![\textsf{loop } B \textsf{ do } S \textsf{ end}]\!]([\![S]\!](\sigma, c)) \\ \qquad\qquad \text{if } \exists u : (active|_u \wedge \sigma(B)|_u) \\ (\sigma, c) \text{ otherwise} \end{cases}$$

If the unfolding does not terminates, then we take the usual convention:

$$[\![\textsf{loop } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = \perp$$

Conditioning. The denotation of a **where** construct is the denotation of its body with a new context. The new context is the conjunction of the previous one with the value of the pure conditioning expression B. If $[\![S]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c')$, then we have

$$[\![\textsf{where } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = (\sigma', c).$$

If $[\![S]\!](\sigma, c \wedge \sigma(B)) = \perp$, then we put $[\![\textsf{where } B \textsf{ do } S \textsf{ end}]\!](\sigma, c) = \perp$. Observe that the value of c' is ignored here, i.e. the initial context is restored on exit from the where block. The evaluation of $\sigma(B)$ is local.

Remark. In the $\mathcal{L}$ language, the activity context is preserved by terminating executions: for any program S such that $[\![S]\!](\sigma, c) = (\sigma', c')$, we have $c = c'$. This will no longer be the case for the extended language below.

3 Extending $\mathcal{L}$ with non-local control transfer commands

The MPL or HyperC languages include the data-parallel extensions of the **for**, **while**, **switch** control structures, with their associated escape commands **break** and **continue**. Their intended meaning is the following. When an escape command is executed at a *currently active* index, the activity at this index is turned to idle until the end of the corresponding enclosing block. We say it is then *asleep*. On the other hand, executing an escape command at an idle index has no effect. Once the control has reached the end of the block, the initial "awakeness" is restored at all indices. This is thus the straightforward data-parallel generalization of the usual scalar behavior, which is to *jump* at the exit label of the block: the jump has been generalized to a *temporary idleness*, which is needed because of the global nature of control.

3.1 Informal description

A simple way to give account of this behavior in the $\mathcal{L}$ language is to extend it with a new block definition structure **begin** S **end**, together with a **escape** command executable in the scope of the block. Such a block will be called an *escaping block*. Yet, this is not sufficient to model the complex interplay between the **break** and **continue** escape commands in C: to do this one would have to be able to escape from more than one enclosing block! The solution is thus to define *several types* of begin_i S **end** escaping blocks with their *respective* escape escape_i commands, labelled $i = 1, \ldots, N$.

The use of escaping blocks is illustrated by the program below. We consider vectors of size 3, where indices range between 0 and 2. The comments at the end of each line show the evolution of the currently active index list. Recall that expression *This* evaluates to the local index value.

```
begin₁                                   /* Indices 0, 1, 2 are active */
    where (This > 0) do                                       /* 1, 2 */
        begin₂                                                /* 1, 2 */
            where (This > 1) do                                  /* 2 */
                escape₁                                       /* None */
            end                                                  /* 1 */
        end                                                      /* 1 */
    end                                                       /* 0, 1 */
end                                                        /* 0, 1, 2 */
```

For usual data-parallel languages derived from C, $N = 2$ is sufficient: type 1 defines for instance the scope of the **break** command, which corresponds to escape_1; type 2 defines the scope of the **continue** command, which corresponds to escape_2. (Note that an additional type could be defined to handle the data-parallel extension of the C **return** command.)

Thanks to these data-parallel non-local transfer commands, our running example can be recast as in Figure 2. On executing the escape_1 command at line 4 at an active index u such that $M|_u = 1$, this index falls asleep, and it remains idle until the end of the enclosing type 1 block at line 8. The loop at line 3 terminates as soon as all indices have turned idle, that is, after all indices have escape_1'd. Once the global control reaches line 8, all asleep indices wake up. The reader may wish to compare this behavior with the usual C code:

$$
\begin{array}{ll}
M := N;\ R := N; & (1) \\
\textsf{begin}_1 & (2) \\
\quad \textsf{loop}\ \mathit{True}\ \textsf{do} & (3) \\
\quad\quad \textsf{where}\ (M = 1)\ \textsf{do escape}_1\ \textsf{end}; & (4) \\
\quad\quad M := M - 1; & (5) \\
\quad\quad R := R \times M & (6) \\
\quad \textsf{end} & (7) \\
\textsf{end} & (8)
\end{array}
$$

Figure 2: The data-parallel factorial, $\mathcal{L}'$ version.

```
m=n; r=n; for (;;) {if (m=1) break; m--; r*=m;}
```

Let $\mathcal{L}'$ be the $\mathcal{L}$ language extended with N block definition structures $\textsf{begin}_i\ S\ \textsf{end}$, $1 \leq i \leq N$, and the N corresponding escape commands $\textsf{escape}_i$.

3.2 A natural semantics for $\mathcal{L}'$

The original notion of *activity* of $\mathcal{L}$ becomes now two-fold in $\mathcal{L}'$:

Conditioning context. It is defined by the where conditioning blocks. An index is said to be *selected* or not according to the value of the conditioning expression.

Escaping context. It is defined for each escaping $\textsf{begin}_i$ block. An index can be *awake* or *asleep*. On executing an $\textsf{escape}_k$ command, all currently active indices fall asleep until the end of the enclosing escaping block of type k.

An index is *active* in $\mathcal{L}'$ if it is both selected with respect to the enclosing conditioning block and, awake with respect to the enclosing escaping blocks of each type. An index not active is said to be *idle*.

The original notion of state (σ, c) in $\mathcal{L}$ can then be similarly extended. A state is now a triple $(\sigma, c, \bar{a})$, where σ is the environment, c is a boolean vector and $\bar{a} = \langle a_1, \ldots, a_N \rangle$ is a list of boolean vectors. Vector c denotes the conditioning context: for each index u, $c|_u$ is true if u is selected, and false otherwise. Each vector a_i denotes the escaping context of type i: for each index u, $a_i|_u$ is true if u is awake in block i, and it is false if it is asleep.

It is convenient to extend the notation defined in Section 2.2.

$$
\mathit{active} \quad \equiv \quad c \wedge \left(\bigwedge_{i=1}^{i=N} a_i\right)
$$

Thanks to this convention, the semantic equations of $\mathcal{L}'$ for assignment, communication, sequencing and iteration are obvious extensions of the corresponding parts of $\mathcal{L}$'s semantics. We only list below the remaining cases.

Conditioning block. On entering a conditioning block, the conditioning context is saved. It is restored on exiting the block. The new conditioning context within the block is the conjunction of the initial one with the current value of the pure expression B.

$$[\![\, \textsf{where } B \textsf{ do } S \textsf{ end} \,]\!](\sigma, c, \bar{a}) = (\sigma', c, \bar{a}')$$

with $[\![\, S \,]\!](\sigma, c \wedge \sigma(B), \bar{a}) = (\sigma', c', \bar{a}')$. Observe that the escaping context is *not* restored. If $[\![\, S \,]\!](\sigma, c \wedge \sigma(B), \bar{a}) = \bot$, then we simply put $[\![\, \textsf{where } B \textsf{ do } S \textsf{ end} \,]\!](\sigma, c, \bar{a}) = \bot$.

Escaping block. On entering an escaping block of type k, the escaping context of type k is saved. It is restored on exiting the block.

$$[\![\, \textsf{begin}_k \; S \textsf{ end} \,]\!](\sigma, c, \bar{a}) = (\sigma', c', \bar{a}')$$

with $[\![\, S \,]\!](\sigma, c, \bar{a}) = (\sigma', c', \bar{a}'')$ and $\bar{a}' = \langle a''_1, \ldots, a''_{k-1}, a_k, a''_{k+1}, \ldots, a''_N \rangle$. if $[\![\, S \,]\!](\sigma, c, \bar{a}) = \bot$, then we put $[\![\, \textsf{begin}_k \; S \textsf{ end} \,]\!](\sigma, c, \bar{a}) = \bot$.

Escape. On executing an $\textsf{escape}_k$ command, all currently *active* indices fall asleep with respect to escaping type k. This amounts to restricting the escaping context of type k with the negation of the current activity context.

$$[\![\, \textsf{escape}_k \,]\!](\sigma, c, \bar{a}) = (\sigma, c, \bar{a}')$$

with $\bar{a}' = \langle a_1, \ldots, a_{k-1}, a_k \wedge \neg active, a_{k+1}, \ldots, a_N \rangle$.

Remark. In the $\mathcal{L}$ language, we have stressed that the activity context is preserved by the terminating executions. Because of the $\textsf{escape}_i$ commands of $\mathcal{L}'$, this invariant is no longer true: an initially active index may be idle at the termination. Yet, it can be proved that an initially idle index remains idle throughout the execution.

4 Assertions and specifications

As for the semantics, we show in this section that the notion of assertion defined for $\mathcal{L}$ programs can be conveniently extended to $\mathcal{L}'$ programs by considering multiple activity contexts. For the sake of completeness, we briefly recall the structure of $\mathcal{L}$ assertions as described in [2].

4.1 An assertion language for $\mathcal{L}$ programs

We define an *assertion language* for the partial correctness of $\mathcal{L}$ programs in the lines of [1]. Such a specification is denoted by a formula $\{Pre\}\ S\ \{Post\}$ where S is the program text, and Pre and $Post$ are logical assertions on variables of S. This formula means that, if precondition Pre is satisfied in the initial state of program S, and if S terminates, then postcondition $Post$ is satisfied in the final state. A proof system gives a formal method to derive such specification formulae by syntax-directed induction on programs. Axioms correspond to statements, and inference rules to control structures. Then, proving that a program meets its specification is equivalent to deriving the specification formula $\{Pre\}\ S\ \{Post\}$ in the proof system.

Such a proof system for the $\mathcal{L}$ language is described in [3]. A fundamental property of this axiomatic semantics in the usual scalar case is *compositionality*. To achieve this goal, the assertion language has to include sufficient information on variable values. Similarly, our assertion language has to include some information about the current activity context as well as variable values. We therefore define two-part assertions $\{P, C\}$, where P is a predicate on vector program variables, and C is a pure boolean vector expression which evaluates into an activity context.

Our assertion language has two kinds of variables, scalar variables and vector variables. As a convention, scalar variables will be denoted with a lowercase initial letter, and vector ones with an uppercase one. We have a similar distinction for arithmetic and logical expressions. As usual, scalar (resp. vector) expressions are inductively defined with usual arithmetic and logical connectives. Basic scalar (resp. vector) expressions are scalar (resp. vector) variables and constants. Vector expression can be subscripted. If the subscript expression is a scalar expression, then we have a scalar expression: the meaning of $X|_u$ is the component of X at index u. Otherwise, if the subscript expression is a vector expression, then we have another vector expression: the meaning of $X|_A$ is a vector whose component at index u is the value of component of X at index $A|_u$. The meaning of a vector expression is obtained by componentwise evaluation. We introduce a scalar conditional expression with a C-like notation $c?e : f$. Its value is the value of expression e if c is true, and f otherwise. Similarly, the value of a conditional vector expression, denoted by $C?E : F$, is a vector whose component at index u is $E|_u$ if $C|_u$ is true, and $F|_u$ otherwise.

Predicates are usual first-order formulae. They are inductively defined from boolean scalar expressions with logical connectives and existential or universal quantifiers binding scalar variables. It turns out that there is no need to consider quantification on vector variables.

We introduce a substitution mechanism for vector variables. Let P be a predicate or any vector expression, X a vector variable, and E a vector expression. $P[E/X]$ denotes the predicate, or expression, obtained by substituting all the occurrences of X in P with E. Note that all vector variables are free by definition of our assertion language. The usual Substitution Lemma [1] extends to this new setting. Let σ be an environment and P a predicate. We use the usual notation $\sigma \models P$ to denote that σ is a model of predicate P, that is, P evaluates to *true* under assignment σ.

Lemma 1 (Substitution lemma) *For every predicate on vector variables P, vector expression E and environment σ,*

$$\sigma \models P[E/X] \quad \textit{iff} \quad \sigma[X \leftarrow \sigma(E)] \models P$$

4.2 Extending assertions to $\mathcal{L}'$ programs

Going from $\mathcal{L}$ to $\mathcal{L}'$ semantics amounts to replacing the single activity context by a conditioning context and a list of escaping contexts. We thus extend the context part of $\mathcal{L}$ assertions in a similar way. Assertions are of the form $\{P, C, \bar{A}\}$, where

- ▹ P is a predicate on program variables;
- ▹ C is a pure boolean vector expression which evaluates into the current conditioning context;
- ▹ $\bar{A} = \langle A_1, \ldots, A_N \rangle$ is a list of pure boolean vector expressions, each A_i evaluates into the current escaping context of type i.

The activity context is the conjunction of these contexts. It is the value of $C \wedge \bigwedge_{i=1}^{i=N} A_i$. For convenience, we denote this expression by $C \wedge \bar{A}$. All definitions of [3] can be extended to this new setting as shown below. We extend the notion of satisfiability (denoted by $\models$) to states and assertions.

Definition 2 (Satisfiability) *Let $(\sigma, c, \bar{a})$ be a state, $\{P, C, \bar{A}\}$ an assertion.*

$$(\sigma, c, \bar{a}) \models \{P, C, \bar{A}\} \qquad \textit{iff} \qquad \sigma \models P \textit{ and } \sigma(C) = c \textit{ and } \forall i : \sigma(A_i) = a_i$$

By convention, $\perp$ satisfies any assertion. The set of states satisfying $\{P, C, \bar{A}\}$ is denoted by $[\![\{P, C, \bar{A}\}]\!]$, or $\{P, C, \bar{A}\}$ when no confusion may arise.

Definition 3 (Assertion implication) *Let $\{P, C, \bar{A}\}$ and $\{Q, D, \bar{B}\}$ be two assertions. We say that the former implies the latter with respect to context,*

$$\{P, C, \bar{A}\} \Rightarrow \{Q, D, \bar{B}\} \qquad \textit{iff} \qquad P \Rightarrow Q \textit{ and } P \Rightarrow \forall u : ((C|_u = D|_u) \wedge \forall i : (A_i|_u = B_i|_u))$$

Observe that this definition extends the usual one: $\{P, C, \bar{A}\} \Rightarrow \{Q, D, \bar{B}\}$ iff $[\![\{P, C, \bar{A}\}]\!] \subseteq [\![\{Q, D, \bar{B}\}]\!]$.

4.3 A proof system for $\mathcal{L}'$ programs

We may now define the validity of a specification of a $\mathcal{L}'$ program with respect to its semantics. Because $\perp$ satisfies any assertion, our notion validity is relative to termination, it defines *partial correctness*.

Definition 4 (Specification validity) *Let S be a $\mathcal{L}'$ program, and let $\{P, C, \bar{A}\}$ and $\{Q, D, \bar{B}\}$ be two assertions. We say that the specification is valid, denoted by*

$$\models \{P, C, \bar{A}\}\ S\ \{Q, D, \bar{B}\}$$

if for each state $(\sigma, c, \bar{a})$ such that $(\sigma, c, \bar{a}) \models \{P, C, \bar{A}\}$

$$[\![S]\!](\sigma, c, \bar{a}) \models \{Q, D, \bar{B}\}.$$

Following the notation of [1], let $\mathit{Change}(S)$ be the set of variables appearing on the left of assignments or as targets of **get** instructions. Only these variables can have their values changed by executing S. Let $\mathit{Var}(C)$ be the set of variables which appear in expression C. The value of C depends on these variables only. We describe below a restricted proof system where we assume everywhere that context expressions are not modified by program bodies: $\mathit{Change}(S) \cap \mathit{Var}(C) = \emptyset$ and $\mathit{Change}(S) \cap (\cup_{i=1}^{i=N} \mathit{Var}(A_i)) = \emptyset$.

Assignment: $X := E$. We extend the usual backwards axiom by taking into consideration that vector variable X is modified only at the active indices, that is indices where $C \wedge \bar{A}$ evaluates to true.

The global activity is preserved by assignments: the initial activity is the same as the final one. As the conditioning and escaping activities are described by boolean vector *expressions*, we can describe the respective initial activities only if the values of the expressions describing the final ones are not changed by the assignment. An easy sufficient condition is that $X \notin \mathit{Var}(C)$ and $\forall i : X \notin \mathit{Var}(A_i)$.

$$\frac{X \notin \mathit{Var}(C) \text{ and } \forall i : X \notin \mathit{Var}(A_i)}{\{P[((C \wedge \bar{A})?E : X)/X], C, \bar{A}\}\ X := E\ \{P, C, \bar{A}\}}$$

Communication: get X from A into Y. As noticed before, a get is an assignment of a remote value.

$$\frac{Y \notin \mathit{Var}(C) \text{ and } \forall i : Y \notin \mathit{Var}(A_i)}{\{P[((C \wedge \bar{A})?X|_A : Y)/Y], C, \bar{A}\}\ \textsf{get}\ X\ \textsf{from}\ A\ \textsf{into}\ Y\ \{P, C, \bar{A}\}}$$

Sequencing: $S; T$. It is a straightforward generalization of the usual case.

$$\frac{\{P, C, \bar{A}\}\ S\ \{R, C', \bar{A}'\},\ \{R, C', \bar{A}'\}\ T\ \{Q, D, \bar{B}\}}{\{P, C, \bar{A}\}\ S; T\ \{Q, D, \bar{B}\}}$$

Iteration: loop B do S end. The usual loop invariant here must be invariant with respect to both the variable values and each of the activity types.

$$\frac{\{I \wedge \exists u : ((C \wedge \bar{A})|_u \wedge B|_u), C, \bar{A}\}\ S\ \{I, C, \bar{A}\}}{\{I, C, \bar{A}\}\ \textsf{loop}\ B\ \textsf{do}\ S\ \textsf{end}\ \{I \wedge \forall u : ((C \wedge \bar{A})|_u \Rightarrow \neg B|_u), C, \bar{A}\}}$$

Conditioning block: where B do S end. Following the semantics, the initial conditioning context is saved on entering the block and restored on exiting. The conditioning context within the block is the conjunction of the conditioning context expression and the conditioning expression. This is taken into account by and-ing conditioning context expression C with condition expression B, and restoring C on exiting. Yet, this makes sense only if the value of C has been left unchanged. The restriction $\mathit{Change}(S) \cap \mathit{Var}(C) = \emptyset$ is an easy sufficient condition for this to hold.

$$\frac{\{P, C \wedge B, \bar{A}\}\ S\ \{Q, C', \bar{A}'\},\ \mathit{Change}(S) \cap \mathit{Var}(C) = \emptyset}{\{P, C, \bar{A}\}\ \textsf{where}\ B\ \textsf{do}\ S\ \textsf{end}\ \{Q, C, \bar{A}'\}}$$

Escaping block: begin$_k$ S end. Similarly, the initial escaping context of type k is saved on entering an escaping block and restored on exiting. Again, the restriction $\mathit{Change}(S) \cap \mathit{Var}(A_k) = \emptyset$ is sufficient to guarantee that the value of A_k has been left unchanged.

$$\frac{\{P, C, \bar{A}\}\ S\ \{Q, C', \bar{A}''\},\ \mathit{Change}(S) \cap \mathit{Var}(A_k) = \emptyset}{\{P, C, \bar{A}\}\ \textsf{begin}_k\ S\ \textsf{end}\ \{Q, C', \bar{A}'\}}$$

with $\bar{A}' = \langle A''_1, \ldots, A''_{k-1}, A_k, A''_{k+1}, \ldots, A''_N \rangle$

Escape: escape_k. All currently active indices fall asleep with respect to escaping type k. The new escaping context expression of type k is the conjunction of the previous one with the negation of the global activity.

$$\{P, C, \bar{A}\}\ \mathsf{escape}_k\ \{P, C, \bar{A}'\}$$

with $\bar{A}' = \langle A_1, \ldots, A_{k-1}, A_k \wedge \neg(C \wedge \bar{A}), A_{k+1}, \ldots, A_N\rangle$

Consequence rule. Following Definition 3, we can state the consequence rule.

$$\frac{\{P, C, \bar{A}\} \Rightarrow \{P', C, \bar{A}'\} \qquad \{P', C', \bar{A}'\}\ S\ \{Q', D', \bar{B}'\} \qquad \{Q', D', \bar{B}'\} \Rightarrow \{Q, D, \bar{B}\}}{\{P, C, \bar{A}\}\ S\ \{Q, D, \bar{B}\}}$$

This rule allows us to strengthen preconditions, and to weaken postconditions of specifications.

Proposition 1 (Soundness) *This proof system is* sound*: if*

$$\vdash \{P, C, \bar{A}\}\ S\ \{Q, D, \bar{B}\}$$

then

$$\models \{P, C, \bar{A}\}\ S\ \{Q, D, \bar{B}\}.$$

Proof ———— *The proof is done by induction on the structure of S. The cases of the assignment and communication commands are simple consequences of the Substitution Lemma 1 thanks to the restriction $X \notin Var(C)$ and $\forall i : X \notin Var(A_i)$. As an example, we give the proof of the case of escaping block.*

Let $(\sigma, c, \bar{a})$ be a state satisfying $\{P, C, \bar{A}\}$. By definition of the escaping block construct, assume $[\![\, \textbf{begin}_k\ S\ \textbf{end}]\!](\sigma, c, \bar{a}) = (\sigma', c', \bar{a}')$ with $[\![\, S]\!](\sigma, c, \bar{a}) = (\sigma', c', \bar{a}'')$ and $a' = \langle a''_1, \ldots, a''_{k-1}, a_k, a''_{k+1}, \ldots, a_N\rangle$. By assumption, $(\sigma', c', \bar{a}'') \models \{Q, C', \bar{A}''\}$. In particular, $\bar{a}'' = \sigma'(\bar{A}'')$.

As $Change(S) \cap Var(A_k) = \emptyset$, we have $\sigma(A_k) = \sigma'(A_k)$. Thus, $\bar{a}' = \sigma'(\bar{A}')$. We get $(\sigma', c', \bar{a}') \models \{Q, C', \bar{A}'\}$, as wanted. ———— $\square$

Remark. Two additional rules will be introduced in the next section to deal with auxiliary variables in preconditions and in programs.

5 An extended example

We demonstrate this proof system by giving the proof annotation of our running example with assertions, in the manner of [11]. Let P be our original program in Figure 2. Let T denote the constant boolean vector whose components are all true. We aim at proving

$$\{\forall u : (N|_u \geq 1), T, \langle T\rangle\} \quad P \quad \{\forall u : (R|_u = N|_u \times \ldots \times 1), T, \langle T\rangle\}$$

The main step is to define a convenient syntactic loop invariant. Observe that the activity context decreases as iterations go. It is thus necessary to add a *new* auxiliary variable A, which is meant to contain the value of the activity context at each iteration. It is sufficient to set it to true initially and

```
(1)   M := N; R := N;
(2)   begin_1
(2')      A := True;
(3)       loop True do
(4')          where (M = 1) do A := False; escape_1 end;
(5)           M := M − 1;
(6)           R := R × M
(7)       end
(8)   end
```

Figure 3: The data-parallel factorial, $\mathcal{L}'$ version with the auxiliary variable

to set it to false just before executing the escape_1 command: this assignment will then be completed exactly at the currently active indices, that is at the indices bound to fall asleep immediately. This new program P' is displayed on Figure 3. According to this intuition, variable A is false at least at all sleeping indices: $\forall u : (\neg A|_u \Rightarrow (M|_u = 1))$. and the role of line $(4')$

$$\mathsf{where}\ (M = 1)\ \mathsf{do}\ A := \mathit{False};\ \mathsf{escape}_1\ \mathsf{end}$$

is to tune the value of A so that $\forall u : (\neg A|_u \Leftrightarrow (M|_u = 1))$. Thus, at each iteration, the escape_1'd indices are exactly those indices u such that $A|_u$ is false, and the activity of type 1 is described by expression A. A good candidate for an invariant is thus $\{I \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)), T, \langle A \rangle\}$, with

$$I \quad \equiv \quad \forall u : ((R|_u = N|_u \times \ldots \times M|_u) \wedge (M|_u \geq 1))$$

Assume for a while that variable A acts as wanted, and that the following annotation can be derived:

$$\{I \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)), T, \langle A \rangle\}$$
$$\mathsf{where}\ (M = 1)\ \mathsf{do}\ A := \mathit{False};\ \mathsf{escape}_1\ \mathsf{end}$$
$$\{I \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)), T, \langle A \rangle\}$$

Then, it is tedious but easy to check that the annotation for the entire program displayed on Figure 4 is valid.

It remains to prove that the annotation of line $(4')$ is indeed correct. This is the only piece of program where the escaping context is explicitly manipulated. Note that variable A appears *both* in the escaping context expression and in the left part of an assignment. The assignment rule cannot be applied, as explained above. We are thus bound to introduce a *new* auxiliary variable A' in the initial assertion in order to save the initial value of the escaping context. First, we show

$$\{I \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)) \wedge \forall u : (A'|_u = A|_u), T, \langle A' \rangle\}$$
$$\mathsf{where}\ (M = 1)\ \mathsf{do}\ A := \mathit{False};\ \mathsf{escape}_1\ \mathsf{end}$$
$$\{I \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)), T, \langle A \rangle\}$$

The annotation is displayed on Figure 5. The crucial step is to show that $(d) \Rightarrow (e)$, that is, boolean

We use the following definitions:

$$
\begin{aligned}
I \equiv\ & \forall u : ((R|_u = N|_u \times \ldots \times M|_u) \wedge (M|_u \geq 1)) \\
I' \equiv\ & \forall u : (A|_u \Rightarrow (R|_u \times M|_u = N|_u \times \ldots \times M|_u) \wedge \neg A|_u \Rightarrow (R|_u = N|_u \times \ldots \times 1) \\
& \quad \wedge (M|_u \geq 1)) \\
I'' \equiv\ & \forall u : (A|_u \Rightarrow (R|_u \times (M|_u - 1) = N|_u \times \ldots \times (M|_u - 1)) \wedge \neg A|_u \Rightarrow (R|_u = N|_u \times \ldots \times 1) \\
& \quad \wedge (M|_u \geq 1))
\end{aligned}
$$

$$\{\forall u : (N|_u \geq 1), T, \langle T \rangle\} \quad (a)$$

(1) $M := N;\ R := N;$

$$\{I, T, \langle T \rangle\} \quad (b)$$

(2) begin$_1$

$$\{I, T, \langle T \rangle\} \quad (c)$$

(2′) $A := True;$

$$\{I \wedge \forall u : A|_u, T, \langle T \rangle\} \quad (d)$$
$$\{I \wedge \forall u : A|_u, T, \langle A \rangle\} \quad (e)$$
$$\{I \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)) \wedge \forall u : A|_u, T, \langle A \rangle\} \quad (f)$$

(3) loop $True$ do

$$\{I \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)) \wedge \exists u : A|_u, T, \langle A \rangle\} \quad (g)$$

(4′) where $(M = 1)$ do $A := False$; escape$_1$ end;

$$\{I \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)), T, \langle A \rangle\} \quad (h)$$
$$\{I'' \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)), T, \langle A \rangle\} \quad (i)$$

(5) $M := M - 1;$

$$\{I' \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)), T, \langle A \rangle\} \quad (j)$$

(6) $R := R \times M$

$$\{I \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)), T, \langle A \rangle\} \quad (k)$$

(7) end

$$\{I \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)) \wedge \forall u : \neg A|_u, T, \langle A \rangle\} \quad (l)$$
$$\{I \wedge \forall u : (M|_u = 1), T, \langle A \rangle\} \quad (m)$$
$$\{\forall u : (R|_u = N|_u \times \ldots \times 1), T, \langle A \rangle\} \quad (n)$$

(8) end

$$\{\forall u : (R|_u = N|_u \times \ldots \times 1), T, \langle T \rangle\} \quad (o)$$

Figure 4: The annotated data-parallel factorial with an auxiliary variable

We use the same definition as on Figure 4.

$$\{I \wedge \forall u : (\neg A'|_u \Rightarrow (M|_u = 1)) \wedge \forall u : (A'|_u = A|_u), T, \langle A' \rangle\} \quad (a)$$

(1) **where** $M = 1$ **do**

$$\{I \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)) \wedge \forall u : (A'|_u = A|_u), (M = 1), \langle A' \rangle\} \quad (b)$$

(2) $A := \mathit{False}$;

$$\{I \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)) \wedge \forall u : (\neg A'|_u \Rightarrow \neg A|_u), (M = 1), \langle A' \rangle\} \quad (c)$$

(3) **escape**$_1$

$$\{I \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)) \wedge \forall u : (\neg A'|_u \Rightarrow \neg A|_u), (M = 1), \langle (A' \wedge \neg(M = 1)) \rangle\} \quad (d)$$

$$\{I \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)) \wedge \forall u : (\neg A'|_u \Rightarrow \neg A|_u), (M = 1), \langle A \rangle\} \quad (e)$$

$$\{I \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)), (M = 1), \langle A \rangle\} \quad (f)$$

(4) **end**

$$\{I \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)), T, \langle A \rangle\} \quad (g)$$

Figure 5: The annotated inner **where** block of the factorial.

vector expressions $A' \wedge \neg(M = 1)$ and A have the same value as soon as predicate

$$\forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)) \wedge (\neg A|'_u \Rightarrow \neg A|_u)$$

is satisfied. This stems from a simple (but tedious) case analysis on the truth value of $A|_u$.

- ▹ If $A|_u$ is true, then both $\neg(M|_u = 1)$ and $A'|_u$ are true. Thus, $(A' \wedge \neg(M = 1))|_u$ is true.
- ▹ If $A|_u$ is false, then $\neg(M|_u = 1)$ is false. Thus, $(A' \wedge \neg(M = 1))|_u$ is false as well.

Let us now introduce an additional rule to our proof system to get rid of such auxiliary variables.

Auxiliary variable elimination in preconditions. If a variable Aux appears in the precondition only, then it can be substituted by any expression E.

$$\frac{\{P, C, \bar{A}\}\ S\ \{Q, D, \bar{B}\} \qquad \mathit{Aux} \notin \mathit{Var}(S) \cup \mathit{Var}(Q) \cup \mathit{Var}(D) \cup \mathit{Var}(\bar{B})}{\{P[E/\mathit{Aux}], C[E/\mathit{Aux}], \bar{A}[E/\mathit{Aux}]\}\ S\ \{Q, D, \bar{B}\}}$$

It can be shown that this rule is sound. Substituting A ($= E$) for A' ($= \mathit{Aux}$) in the initial precondition on Figure 5 yields the wanted formula:

$$\{I \wedge \forall u : (\neg A|_u \Rightarrow (M|_u = 1)), T, \langle A \rangle\}$$
$$\textsf{where } (M = 1) \textsf{ do } A := \mathit{False};\ \textsf{escape}_1 \textsf{ end}$$
$$\{I \wedge \forall u : (\neg A|_u \Leftrightarrow (M|_u = 1)), T, \langle A \rangle\}$$

It remains to get rid of auxiliary variable A in the factorial program P'. We can again add a new rule in our proof system, which enables to forget everything about such auxiliary variables, in the lines of the method proposed by Gries and Owicki [11].

Definition 5 (Auxiliary variables) *Let V be a set of variables. We say that variables of V are* auxiliary *in program S if they only appear in assignment commands of the form $Z := E$, or communication commands of the form* **get** *X* **from** *A* **into** *Z, with $Z \in V$.*

It is clear that removing all commands containing variables of V does not modify the overall behavior of the program, nor the final values of the variables not in V. The role of such auxiliary variables is limited to convey information from the control flow and the activity context to the environment. If S is a program, and V is a set of auxiliary variables for S, then $S \setminus V$ denotes the program obtained by stripping S from all commands involving variables of V.

Elimination of auxiliary variables in programs. If something which does not depend on auxiliary variables has been proved about a program equipped with auxiliary variables, then it is true of the program without them.

$$\frac{\begin{array}{c}\{P, C, \bar{A}\}\ S\ \{Q, D, \bar{B}\} \\ V \text{ is a set of auxiliary variables for } S \\ V \cap (\mathit{Var}(P) \cup \mathit{Var}(C) \cup \mathit{Var}(\bar{A}) \cup \mathit{Var}(Q) \cup \mathit{Var}(D) \cup \mathit{Var}(\bar{B})) = \emptyset\end{array}}{\{P, C, \bar{A}\}\ S \setminus V\ \{Q, D, \bar{B}\}}$$

It can be shown that this rule is sound. It is clear that $\{A\}$ is a set of auxiliary variables for P', and that $P' \setminus V$ is exactly P. From the proved formula

$$\{\forall u : (N|_u \geq 1), T, \langle T \rangle\} \quad P' \quad \{\forall u : (R|_u = N|_u \times \ldots \times 1), T, \langle T \rangle\}$$

we can finally infer the desired formula:

$$\{\forall u : (N|_u \geq 1), T, \langle T \rangle\} \quad P \quad \{\forall u : (R|_u = N|_u \times \ldots \times 1), T, \langle T \rangle\}$$

6 Conclusion

This work shows that the classical approach towards the natural semantics and assertional proof systems for scalar languages can be extended to modern data-parallel languages. It can even be tuned to handle complex escape control structures as found in high-level data-parallel languages such as MasPar's MPL or the recent HyperC. Our running example shows that the proof of such programs can be built according to the usual intuition by annotating the program text with intermediate assertions. This is a major result, as the amount of information is much larger than in the scalar case, and yet the formal manipulations are basically of the same complexity.

To our understanding, this is a strong argument in favor of data-parallel programming as opposed to (control-)parallel Occam-like programming: data-parallelism allows to handle the validation of parallel programs "for free", which is in striking contrast to the technical complexity of the validation methods for Occam programs.

This work can be continued in several directions. On a technical level, it would be interesting to study the completeness of the proof system (at least for programs without iteration): is it always possible to add auxiliary variables to convey enough information from the control flow to the environment? Also, we have shown in [4] that any $\mathcal{L}'$ program S can be transformed into an equivalent $\mathcal{L}$ program S', up to auxiliary variables. In [3], we have presented a proof system for $\mathcal{L}$ programs. What is the relationship between the proof of S in $\mathcal{L}'$, and the proof of the equivalent program S' in $\mathcal{L}$? Also, the MPL and HyperC languages do not include the escape mechanism explicitly, but rather through the specialized **break** and **continue** commands: can we define any specialized proof rules to handle these constructs directly?

On a broader level, the extension of usual proof systems to complex data-parallel languages such as MPL or HyperC, enables to reuse in this new setting all the know-how developed for the validation of scalar programs: methodologies, computer-aided verification environments (e.g., [8]), heuristics, etc. This opportunity opens a quite exciting research direction which could make large-scale parallel programming *really* possible. We are currently investigating this new frontier.

Acknowledgments. This work has greatly benefited from discussions with Yann Le Guyadec and Bernard Virot. We thank Gaétan Hains for his detailed comments and suggestions.

References

[1] K.R. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs.* Text and Monographs in Computer Science. Springer Verlag, 1990.

[2] L. Bougé, Y. Le Guyadec, G. Utard, and B. Virot. On the expressivity of a waekest precondition calculus for a simple data-parallel language. In *Parallel Processing: ConPar'94 – VAPP VI*, Lect. Notes Comp. Science, Linz, Austria, September 1994.

[3] L. Bougé, Y. Le Guyadec, G. Utard, and B. Virot. A proof system for a simple data-parallel programming language. In C. Girault, editor, *Proc. of the IFIP WG10.3 Int. Conf. on Application in Parallel and Distributed Computing*, Caracas, Vénézuela, April 1994. Elsevier.

[4] L. Bougé and J.-L. Levaire. Control structures for data-parallel SIMD languages: semantics and implementation. *FGCS*, 8:363–378, 1992.

[5] M. Clint and K.T. Narayana. On the completeness of a proof system for a synchronous parallel programming langage. In *Third Conf. Found. Softw. Techn. and Theor. Comp. Science*, Bangalore, India, December 1983.

[6] High Performance Fortran Forum. *High Performance Fortran language specification (draft version).* CITI/CRPC, Rice Univ., Houston, January 1993. Version 1.0 Draft.

[7] J. Gabarró and R. Gavaldà. An approach to correctness of data parallel algorithms. Technical Report LSI-91-19, Univ. Politécnica de Catalunya, October 1991. To appear in Journ. of Parallel and Distr. Computing, 1994.

[8] M.J.C. Gordon. *Programming Language Theory and its Implementation.* Int. Series in Comp. Sciences. Prentice Hall, 1988.

[9] J.-L. Levaire. Using the Centaur system for data-parallel SIMD programming: a case study. In *Proc. 4th European Symposium on Programming, ESOP'92*, volume 582 of *Lect. Notes Comp. Science*, pages 341–350. Springer-Verlag, February 1992.

[10] MasPar Computer Corporation, Sunnyvale CA. *Maspar Parallel Application Language Reference Manual*, 1990.

[11] S. Owicki and D. Gries. Verifying Properties of Parallel Programs : An Axiomatic Approach. *Communication of the ACM*, 19(5):279–285, May 1976.

[12] N. Paris. HyperC specification document. Technical Report 93-1, HyperParallel Technologie, 1993.

[13] R.H. Perrot. A language for array and vector processors. *ACM Trans. on Programming languages and Systems*, 1(2):177–195, 79.

[14] A. Stewart. An axiomatic treatment of SIMD assignment. *Bit*, 30:70–82, 1990.

[15] Thinking Machine Corporation, Cambridge MA. *C^* programming guide*, 1990.