On the completeness of a proof system for a simple data-parallel programming language (extended abstract)*

Luc Bougé **, David Cachera

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

Abstract. We prove the completeness of an assertional proof system for a simple loop-free data-parallel language. This proof system is based on two-part assertions, where the predicate on the current value of variables is separated from the specification of the current extent of parallelism. The proof is based on a Weakest Precondition (WP) calculus. In contrast with the case of usual scalar languages, not all WP can be defined by an assertion. Yet, partial definability suffices to prove the completeness thanks to the introduction of hidden variables in assertions. The case of data-parallel programs with loops is briefly discussed in the conclusion.

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

1 Introduction

The development of massively parallel computing in the last two decades has called for the elaboration of a parallel programming model. The data-parallel programming model has proven to be a good framework, since it allows the easy development of applications portable across a wide variety of parallel architectures. The increasing role of this model requires appropriate theoretical foundations. These foundations are crucial to design safe and optimized compilers, and programming environments including parallelizing, data-distributing and debugging tools. They are also the way to safe programming techniques, so as to avoid the common waste of time and money spent in debugging.

Existing data-parallel languages, such as HPF, C*, HyperC or MPL, include a similar core of data-parallel control structures. In previous papers, we have shown that it is possible to define a simple but representative data-parallel kernel language (the $\mathcal{L}$ language), to give it a formal operational [5] and denotational semantics [4], and to define a proof system for this language, in the style of the usual Hoare's logic approach [10, 4]. The originality of our approach lies in the treatment of the *extent of parallelism*, that is, the subset of currently active

* The full version of this paper can be found in [2].

** Authors contact: Luc Bougé (Luc.Bouge@lip.ens-lyon.fr). This work has been partly supported by the French CNRS Research Program on Parallelism, Networks and Systems(PRS).

indices at which a vector instruction is to be applied. Previous approaches led to manipulate lists of indices explicitly [6, 11], or to consider context expressions as assertions modifiers [8]. In contrast, our proof system for $\mathcal{L}$ describes the activity context by a vector boolean expression distinct from the usual predicates on program variables.

We have shown that our proof system for $\mathcal{L}$ is *sound*, that is, any provable property of a program is actually valid. In this paper, we address the converse *completeness* problem: *Can any valid property of a program be proved in our system?* In some sense, completeness guarantees that the rules of a proof system actually catch all the semantic expressiveness of the language under study.

The completeness of proof systems for scalar Pascal-like languages has been extensively studied [1]. In attacking such a problem, the main tool is the *weakest preconditions calculus*. This notion has been introduced by Dijkstra [7]. It plays a central role in the formal validation of scalar programs, as shown in [9] for instance. The case of data-parallel programs is much more complex than the case of scalar programs, as one has to cope *both* with the variable values and with the manipulations of the activity context. Yet, we have shown in [3] that it is possible to define a weakest preconditions calculus for $\mathcal{L}$, at least for loop-free (so-called straight-line or *linear*) programs.

The contribution of this paper is to apply these results to prove the completeness of our proof system for *all linear* programs. We proceed as follows. We first present the $\mathcal{L}$ language, and give its denotational semantics. We describe a sound assertional proof system for this language, as defined in [4], and its weakest preconditions calculus as described in [3]. Then, we prove the completeness of the proof system in a restricted case: *plain* specifications formulae and *regular* programs. To handle non-regular programs, we extend the proof system with an additional rule. It enables to introduce and eliminate auxiliary *hidden* variables in assertions. We prove that this extended proof system is complete for linear programs without any restriction.

2 The $\mathcal{L}$ Language

An extensive presentation of the $\mathcal{L}$ language can be found in [5]. For the sake of completeness (if we dare say so!), we briefly recall its denotational 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: *component-wise* operations, or 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*. 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*, HYPERC or MPL. We do not consider the scalar part of these languages, mainly imported from the *C* language. For the sake of simplicity, we consider a

unique geometry of arrays: arrays of dimension one, also called *vectors*. Then, all the 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 component of parallel object X located at index u is denoted by $X|_u$. The legal expressions are usual *pure* expressions: the value of a pure expression at index u only depends on the values of the variables components at index u. The expressions are evaluated by applying operators *component-wise* to parallel values. We do not specify the syntax and semantics of such expressions any further. A particular vector expression is called *This*. The value of its component at each index u is the value u itself: $\forall u : This|_u = u$. Note that *This* is a pure expression and that all constructs defined here are *deterministic*. The set of $\mathcal{L}$-instructions is the following.

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.

Sequencing: $S;T$. On the termination of the last action of S, the execution of the actions of T starts.

Iteration: loop B do S. 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. The current activity context is not modified.

Conditioning: where B do S. The active indices where *pure* boolean expression B evaluates to false become idle during the execution of S. The other ones remain active. The initial activity context is restored on the termination of S.

2.2 Denotational Semantics of $\mathcal{L}$

We recall the semantics of $\mathcal{L}$ defined in [3] in the style of denotational 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 the parallel expressions: $\sigma(E)$ denotes the value obtained by evaluating parallel expression E in environment σ. We do not detail the internals of expressions any further. Note that $\sigma(This)|_u = u$ 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. The set of contexts is denoted by Ctx. We distinguish a particular context denoted by *True* where all components have value *true*. The context *False* is defined the same way. For convenience, we define the activity predicate $Active_c$: $Active_c(u) \equiv c|_u$.

A *state* is a pair made of an environment and a context. The set of states is denoted by *State*: $State = (Env \times Ctx) \cup \{\bot\}$ where $\bot$ denotes the undefined state. The semantics $[\![S]\!]$ of a program S is a *strict* function from *State* to *State*. $[\![S]\!](\bot) = \bot$, and $[\![S]\!]$ is extended to sets of states as usual.

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 $Active_c(u)$, and $V|_u = \sigma(X)|_u$ otherwise. The activity context is preserved.

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 $Active_c(u)$, and $V|_u = \sigma(Y)|_u$ otherwise.

Sequencing: Sequential composition is 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.

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

If the unfolding does not terminate, then we take the usual convention: $[\![\textsf{loop } B \textsf{ do } S]\!](\sigma, c) = \bot$.

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.

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

with $[\![S]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c')$.
If $[\![S]\!](\sigma, c \wedge \sigma(B)) = \bot$, then we put $[\![\textsf{where } B \textsf{ do } S]\!](\sigma, c) = \bot$. Observe that the value of c' is ignored here.

3 A Sound Assertional Proof System for the $\mathcal{L}$ Language

3.1 Assertion Language

We define an *assertion language* for the correctness of $\mathcal{L}$ programs in the lines of [1]. Such a specification is denoted by a formula $\{P\}\ S\ \{Q\}$ where S is the program text, and P and Q are two logical assertions on the variables of S. This formula means that, if precondition P is satisfied in the initial state of program S, and if S terminates, then postcondition Q is satisfied in the final state. A proof system gives a formal method to derive such specification formulae by syntax-directed induction on programs.

We recall below the proof system described in [3]. As in the usual sequential case, the assertion language must be powerful enough to express properties on variable values. Moreover, it has to handle the evolution of the activity context

along the execution. An assertion shall thus be broken up into two parts: $\{P, C\}$, where P is a predicate on program variables, and C a pure boolean vector expression. The intuition is that the current activity context is exactly the value of C in the current state, as expressed in the definition below.

Definition 2 (Satisfiability). Let (σ, c) be a state, and $\{P, C\}$ an assertion. We say that (σ, c) *satisfies* the assertion $\{P, C\}$, denoted by $(\sigma, c) \models \{P, C\}$, if $\sigma \models P$ and $\sigma(C) = c$. By convention, $\perp$ satisfies any assertion. The set of states satisfying $\{P, C\}$ is denoted by $[\![\{P, C\}]\!]$. When no confusion may arise, we identify $\{P, C\}$ and $[\![\{P, C\}]\!]$.

Observe that there are many sets of states which cannot be described by any assertion. This is in strong contrast with the case of scalar Pascal-like languages.

Lemma 3 (Restricted power of assertions). *Let $\{P, C\}$ an assertion. For any environment σ, there exists at most* one *activity context c such that $(\sigma, c) \in [\![\{P, C\}]\!]$, namely $c = \sigma(C)$.*

So, if a set of states contains two states (σ, c) and (σ, c') with $c \neq c'$, then it cannot be described by any assertion.

Definition 4 (Assertion implication). Let $\{P, C\}$ and $\{Q, D\}$ be two assertions. We say that $\{P, C\}$ implies $\{Q, D\}$, and write $\{P, C\} \Rightarrow \{Q, D\}$, iff

$$(P \Rightarrow Q) \quad \text{and} \quad (P \Rightarrow \forall u : (C|_u = D|_u))$$

Proposition 5. *Let $\{P, C\}$ and $\{Q, D\}$ be two assertions. Then*

$$\{P, C\} \Rightarrow \{Q, D\} \quad \textit{iff} \quad [\![\{P, C\}]\!] \subseteq [\![\{Q, D\}]\!]$$

Our assertion language manipulates two kinds of variables, *scalar* variables and *vector* variables. As a convention, scalar variables are denoted with a lowercase initial letter, and vector ones with an uppercase one. We have a similar distinction on arithmetic and logical expressions. As usual, scalar (resp. vector) expressions are recursively 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. Otherwise, if the subscript expression is a vector expression, then we have another vector expression. The meaning of a vector expression is obtained by component-wise 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 recursively defined on boolean scalar expressions with logical connectives and existential and universal quantifiers on scalar variables. Note that we do not 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.

Lemma 6 (Substitution Lemma). *For every predicate on vector variables P, vector expression E and environment σ,*

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

We can define the validity of a specification of a $\mathcal{L}$ program with respect to its denotational semantics.

Definition 7 (Specification validity). Let S be a $\mathcal{L}$ program, $\{P, C\}$ and $\{Q, D\}$ two assertions. We say that specification $\{P, C\}\ S\ \{Q, D\}$ is valid, denoted by $\models \{P, C\}\ S\ \{Q, D\}$, if for all states (σ, c)

$$((\sigma, c) \models \{P, C\}) \Rightarrow ([\![S]\!](\sigma, c) \models \{Q, D\}).$$

Since $\perp$ satisfies any assertion, validity is relative to partial correctness.

3.2 Proof System

We recall on Fig. 1 the proof system defined in [3]. This system is a restricted proof system, in the sense that it only manipulates a certain kind of specification formulae, precisely these formulae $\{P, C\}\ S\ \{Q, D\}$ such that the boolean vector expression D describing the final activity context may not be modified by the program S. More formally, using the notations of [1], we define the following sets of variables.

Definition 8. Let E be an expression. $Var(E)$ is the set of all variables appearing in E. Expression E may only depend on the values of these variables. We extend this definition to a $\mathcal{L}$-program S: $Var(S)$ is the set of all variables appearing in S.

Let S be a $\mathcal{L}$-program. $Change(S)$ is the set of program variables which appear on the left-hand side of an assignment statement or as the target of a communication statement. Only these variables may be modified by executing S.

A sufficient condition to guarantee the absence of interference between S and D is thus $Change(S) \cap Var(D) = \emptyset$. If a specification formula $\{P, C\}\ S\ \{Q, D\}$ is derivable in the proof system, then we write $\vdash \{P, C\}\ S\ \{Q, D\}$.

Theorem 9 (Soundness of $\vdash$ [4]). *The $\vdash$ proof system is sound: If $\vdash \{P, C\}\ S\ \{Q, D\}$, then $\models \{P, C\}\ S\ \{Q, D\}$.*

The contribution of this paper is to address the converse problem:

> *Is any valid specification provable? Does $\models \{P, C\}\ S\ \{Q, D\}$ imply $\vdash \{P, C\}\ S\ \{Q, D\}$?*

In the case of scalar programs, the completeness of the proof systems for loop-free (so-called *linear*) programs is a not-so-difficult consequence of the existence of a weakest preconditions calculus. In contrast, studying the completeness with respect to programs including loops requires sophisticated methods. Indeed, this is tightly connected to the expressivity of the underlying logic.

In the case of data-parallel programs, even the simplest case of loop-free programs is already non-trivial. In this paper, we therefore restrict ourselves to this case. The case of loops is discussed in the conclusion.

Definition 10 (Linear $\mathcal{L}$-programs). A data-parallel $\mathcal{L}$-program S is *linear* if it is made of assignments, communications, sequencing and conditioning only.

Assignment Rule	$\dfrac{X \notin Var(D)}{\{Q[(D?E : X)/X], D\}\ X{:=}E\ \{Q, D\}}$
Communication Rule	$\dfrac{Y \notin Var(D)}{\{Q[(D?X\|_A : Y)/Y], D\}\ \mathsf{get}\ X\ \mathsf{from}\ A\ \mathsf{into}\ Y\ \{Q, D\}}$
Sequencing Rule	$\dfrac{\{P, C\}\ S\ \{R, E\},\ \{R, E\}\ T\ \{Q, D\}}{\{P, C\}\ S;T\ \{Q, D\}}$
Conditioning Rule	$\dfrac{\{P, C \wedge B\}\ S\ \{Q, D\},\ Change(S) \cap Var(C) = \emptyset}{\{P, C\}\ \mathsf{where}\ B\ \mathsf{do}\ S\ \{Q, C\}}$
Consequence Rule	$\dfrac{\{P, C\} \Rightarrow \{P', C'\},\ \{P', C'\}\ S\ \{Q', D'\},\ \{Q', D'\} \Rightarrow \{Q, D\}}{\{P, C\}\ S\ \{Q, D\}}$
Iteration Rule	$\dfrac{\{I \wedge \exists u : (C\|_u \wedge B\|_u), C\}\ S\ \{I, C\}}{\{I, C\}\ \mathsf{loop}\ B\ \mathsf{do}\ S\ \{I \wedge \forall u : (C\|_u \Rightarrow \neg B\|_u), C\}}$

Fig. 1. The $\vdash$ proof system for $\mathcal{L}$

4 Completeness of the Proof System for Plain Specifications and Regular, Linear Programs

4.1 Weakest Preconditions Calculus

Our main tool to prove the completeness of our system is a weakest preconditions calculus. This calculus has been presented in [3], and we briefly recall the main results below. Let us first motivate its use. We want to demonstrate that

$$\models \{P, C\}\ S\ \{Q, D\} \quad \Rightarrow \quad \vdash \{P, C\}\ S\ \{Q, D\}.$$

Let $\{P, C\}\ S\ \{Q, D\}$ be a valid specification formula. Assume for a while that we can find an assertion $\{P', C'\}$ such that $\vdash \{P', C'\}\ S\ \{Q, D\}$ holds, and moreover $\{P, C\} \Rightarrow \{P', C'\}$. Then, using the Consequence Rule, we have demonstrated

$$\vdash \{P, C\}\ S\ \{Q, D\}.$$

Definition 11 (Weakest preconditions). Let S be a linear $\mathcal{L}$-program and $\{Q, D\}$ an assertion. We define the *weakest preconditions* as

$$wp(S, \{Q, D\}) = \{s \in State \mid [\![S]\!](s) \models \{Q, D\}\}$$

Observe that

$$\models \{P, C\}\ S\ \{Q, D\} \quad \Leftrightarrow \quad [\![\{P, C\}]\!] \subseteq wp(S, \{Q, D\}).$$

As we have restricted ourselves to linear programs, observe that we do not need to take into account problems of divergence.

We have shown in a previous paper [3] that the set of states $wp(S, \{Q, D\})$ cannot generally be described by some assertion $\{P, C\}$ and thus be manipulated in the proof system. It is only the case when certain syntactic non-interference conditions on S and D are satisfied. These conditions are summed up in the following definitions.

Definition 12 (Plain specification). A specification formula $\{P, C\}\ S\ \{Q, D\}$ is said to be *plain* if we have $Var(D) \cap Change(S) = \emptyset$.

Definition 13 (Regular program). A program P is *regular* if, for any subprogram of P of the form where B do S, we have $Var(B) \cap Change(S) = \emptyset$.

Observe that any subprogram of a regular program is regular, too. The detailed definability results are listed up on Fig. 2. In spite of the restrictions, they are sufficient to guarantee the following property.

Proposition 14 (Restricted definability of wp for regular programs [3]). *Let S be a regular, linear $\mathcal{L}$-program, and let $\{\ldots\}\ S\ \{Q, D\}$ be a plain specification for S. Then there exists an assertion $\{P, D\}$ such that $[\![\{P, D\}]\!] = wp(S, \{Q, D\})$ In particular, $\models \{P, D\}\ S\ \{Q, D\}$.*

Construct	Conditions	Weakest Precondition
Assignment	$X \notin Var(D)$	$wp(X := E, \{Q, D\})$ $= \{Q[(D?E : X)/X], D\}$
Communication	$Y \notin Var(D)$	$wp(\textsf{get } X \textsf{ from } A \textsf{ into } Y, \{Q, D\})$ $= \{Q[(D?X\|_A : Y)/Y], D\}$
Sequencing	—	$wp(S_1;S_2, \{Q, D\})$ $= wp(S_1, wp(S_2, \{Q, D\}))$
Conditioning	$(Var(D) \cup Var(B)) \cap Change(S)$ $= \emptyset$ $wp(S, \{Q, D \wedge B\}) = \{P, C\}$	$wp(\textsf{where } B \textsf{ do } S \textsf{ end}, \{Q, D\})$ $= \{P, D\}$

Fig. 2. Definability properties of weakest preconditions for regular, linear $\mathcal{L}$-programs

4.2 Proving the Restricted Completeness

We now want to establish the completeness for the proof system described on Fig. 1. As it concerns plain specifications and regular programs, we call it *restricted* completeness. More formally, we aim at proving the following theorem.

Theorem 15 (Restr. compl., plain specif., regular, linear progr.). *Let $\{P, C\}\ S\ \{Q, D\}$ be a plain specification, with S a regular, linear program.*

$$\textit{If } \models \{P, C\}\ S\ \{Q, D\}, \quad \textit{then } \vdash \{P, C\}\ S\ \{Q, D\}.$$

Proof. The proof of this theorem follows the lines of [1]. It uses the weakest preconditions calculus. For any regular, linear program S and any plain specification $\{P, C\}\ S\ \{Q, D\}$, there exists some assertion $\{P', C'\}$ such that

$[\![\{P', C'\}]\!] = wp(S, \{Q, D\})$. Using the Consequence Rule, it suffices to demonstrate that

$$\vdash wp(S, \{Q, D\})\ S\ \{Q, D\}.$$

The proof is by induction on the structure of the regular, linear program S, using the definability properties of Fig. 2.

The cases of the assignment, communication, and sequencing constructs are straightforward. Let us consider the case of the conditioning construct, with $S \equiv$ **where** B **do** T. As S is regular by hypothesis, we have $Change(T) \cap Var(B) = \emptyset$. As the specification is plain, we have $Change(S) \cap Var(D) = \emptyset$. As $Change(T) = Change(S)$, we also have $Change(T) \cap Var(D) = \emptyset$. The Definability Property 14 yields an assertion $\{P, D \wedge B\}$ such that $\{P, D \wedge B\} = wp(T, \{Q, D \wedge B\})$.

Program T is regular and linear as S is so. Specification $\{P, D \wedge B\}\ T\ \{Q, D \wedge B\}$ is plain.

Thus, the induction hypothesis yields $\vdash \{P, D \wedge B\}\ T\ \{Q, D \wedge B\}$. As $(Var(B) \cup Var(D)) \cap Change(T) = \emptyset$, the **where** Rule of the proof system applies, and we get $\vdash \{P, D\}$ **where** B **do** $T\ \{Q, D\}$.

Furthermore, the Definability Property gives $wp($**where** B **do** $T, \{Q, D\}) = \{P, D\}$. Hence the desired result:

$$\vdash wp(\textbf{where } B \textbf{ do } T, \{Q, D\})\ \textbf{where } B \textbf{ do } T\ \{Q, D\}.$$

□

5 Extending the Proof of Completeness to Non-regular, Linear Programs

We have demonstrated the completeness of the proof system for plain specifications and regular, linear programs. In the presence of non-regular programs, we are no longer able to find any assertion that expresses the weakest preconditions. Thus, we first have to transform a non-regular program into a regular one. This can be done by introducing an *auxiliary variable*, which stores the value of the vector boolean expression: program **where** B **do** S is transformed into $Tmp := B$; **where** Tmp **do** S

Using such a variable can be interpreted as keeping track of the nested activity context in a stack. Each new variable Tmp is a cell of the stack.

But, instead of transforming programs in order to be able to prove them, we claim that it is possible *to encapsulate* this transformation into the proof system itself. The notion corresponding to syntactic *auxiliary* variables in programs is that of semantic *hidden* variables in assertions.

Rule 1 (Elimination of hidden variables) *Let E be any vector expression.*

$$\frac{\{P, C\}\ S\ \{Q, D\}, \quad Tmp \notin Var(S) \cup Var(Q) \cup Var(D)}{\{P[E/Tmp], C[E/Tmp]\}\ S\ \{Q, D\}}$$

We denote by $\vdash^* \{P, C\}\ S\ \{Q, D\}$ that a specification formula is derivable in the $\vdash$ proof system augmented with this new rule.

Theorem 16 (Soundness of $\vdash^*$). *The $\vdash^*$ proof system is sound:*

$$\textit{If } \vdash^* \{P\}\ S\ \{Q\}, \quad \textit{then } \models \{P\}\ S\ \{Q\}$$

Proof. Easy, using the Substitution Lemma — see [2]. □

Theorem 17 (Restr. completeness, plain specif., linear program). *Let* $\{P, C\}\ S\ \{Q, D\}$ *be a plain specification, with* S *a linear program.*

$$\textit{If } \models \{P, C\}\ S\ \{Q, D\}, \quad \textit{then } \vdash^* \{P, C\}\ S\ \{Q, D\}$$

We first state that, thanks to the introduction of hidden variables, we retain the properties of definability of the weakest preconditions. The following proposition guarantees the existence of some assertion describing the weakest preconditions of any conditioning construct.

Proposition 18 (Non-regular conditioning [3]). *Let* $Tmp \notin Var(S) \cup Var(Q) \cup Var(D)$. *If*

$$wp(S, \{Q, D \wedge Tmp\}) = \{P, C\},$$

then

$$wp(\textsf{where } B \textsf{ do } S, \{Q, D\}) = \{P[B/Tmp], D\}$$

Theorem 19 (Restricted definability of WP). *Let* S *be a linear program. Let* $\{\ldots\}\ S\ \{Q, D\}$ *be a plain specification for* S. *Then there exists an assertion* $\{P, D\}$ *such that* $[\![\{P, D\}]\!] = wp(S, \{Q, D\})$. *In particular,* $\models \{P, D\}\ S\ \{Q, D\}$.

We can now prove Completeness Theorem 17 for non-regular programs.

Proof. The proof is similar to that of the Completeness Theorem 15 for regular programs. It uses a structural induction on S. The only new case to consider is $S \equiv \textsf{where } B \textsf{ do } T$, with $Var(B) \cap Change(T) \neq \emptyset$. Pick up a "new" variable Tmp such that $Tmp \notin Var(S) \cup Var(Q) \cup Var(D)$. Such a variable exists because the expressions from the program and from the assertion language are finite terms. By Theorem 19, we know there exists some assertion $\{P, D \wedge Tmp\} = wp(T, \{Q, D \wedge Tmp\})$.

By the induction hypothesis, we have $\vdash^* \{P, D \wedge Tmp\}\ T\ \{Q, D \wedge Tmp\}$. We also have $\{P \wedge B = Tmp, D \wedge B\} \Rightarrow \{P, D \wedge Tmp\}$.

We can thus apply the Consequence Rule. This yields $\vdash^* \{P \wedge B = Tmp, D \wedge B\}\ T\ \{Q, D \wedge Tmp\}$. Then, we apply the **where** Rule, and we get $\vdash^* \{P \wedge B = Tmp, D\}$ **where** B **do** $T\ \{Q, D\}$. Thanks to the Consequence Rule, this rewrites into $\vdash^* \{P[B/Tmp] \wedge B = Tmp, D\}$ **where** B **do** $T\ \{Q, D\}$.

Finally, applying the Elimination Rule with $E \equiv B$ yields $\vdash^* \{P[B/Tmp], D\}$ **where** B **do** $T\ \{Q, D\}$. According to Proposition 18, $wp(S, \{Q, D\}) = \{P[B/Tmp], D\}$. Thus $\vdash^* wp(S, \{Q, D\})\ S\ \{Q, D\}$. As before, we conclude the proof with the Consequence Rule and the Definability Property 19. □

6 Extending the Proof of Completeness to Non-plain Specifications

We now focus on general specifications, where $Var(D) \cap Change(S)$ may be not empty. Surprisingly enough, the Elimination Rule is sufficient to prove the completeness in this case, and there is no need of any other additional rule.

Theorem 20 (Completeness, linear programs). *Let S be a linear program.*

$$\textit{If } \models \{P, C\}\ S\ \{Q, D\}, \quad \textit{then } \vdash^* \{P, C\}\ S\ \{Q, D\}.$$

Proof. Assume $\models \{P, C\}\ S\ \{Q, D\}$. As the expressions of the assertion language are finite terms, there exists a "new" hidden variable Tmp such that $Tmp \notin Var(S) \cup Var(Q) \cup Var(D)$. Let us show that

$$\models \{P \wedge Tmp = C, C\}\ S\ \{Q \wedge Tmp = D, Tmp\}$$

Let (σ, c) be in $[\![\{P \wedge Tmp = C, C\}]\!]$. We have in particular $(\sigma, c) \models \{P, C\}$. By hypothesis, we get $[\![S]\!](\sigma, c) = (\sigma', c) \models \{Q, D\}$.

Furthermore, we have $\sigma(Tmp) = \sigma(C) = c$. As $Tmp \notin Var(S)$, we have $\sigma'(Tmp) = \sigma(Tmp) = c$, and $(\sigma', c) \models \{Q, D\}$ gives $\sigma'(D) = c$. We conclude that $(\sigma', c) \models \{Q \wedge Tmp = D, Tmp\}$.

As $Tmp \notin Var(S)$, we are in the case of a plain specification, so the Completeness Theorem 17 applies and yields $\vdash^* \{P \wedge Tmp = C, C\}\ S\ \{Q \wedge Tmp = D, Tmp\}$.

As $\{Q \wedge Tmp = D, Tmp\} \Rightarrow \{Q, D\}$, we can apply the Consequence Rule. It yields $\vdash^* \{P \wedge Tmp = C, C\}\ S\ \{Q, D\}$. Applying then the Elimination Rule with $E \equiv C$ yields $\vdash^* \{P \wedge C = C, C\}\ S\ \{Q, D\}$.

Finally, as $\{P, C\} \Rightarrow \{P \wedge C = C, C\}$, we deduce by another application of the Consequence Rule that

$$\vdash^* \{P, C\}\ S\ \{Q, D\}$$

□

7 Conclusion

We have proved the completeness of a proof system associated with a simple data-parallel programming language. This proof system is based on a two-part assertion language, which enables a convenient treatment of activity context specifications.

We restricted ourselves to loop-free (so-called *linear*) programs. The proof of completeness given here relies on a weakest preconditions calculus, as in similar proofs for usual scalar (sequential) languages. The main technical difficulty is to cope with context manipulations. We first established restricted results of completeness, assuming syntactic restrictions on the conditioning constructs and on postconditions in the specification formulae. In a second step, we introduced a notion of *hidden* variables, together with an additional proof rule to manipulate them. We could then establish that this augmented proof system is complete for unrestricted programs and specifications formulae.

This completeness result can be extended to programs with loops using techniques similar to the usual scalar cases. Observe that handling loops in the $\mathcal{L}$ language is a subtle task, as the loop construct of the $\mathcal{L}$ language introduces a global logical OR on *infinite* boolean vectors. To illustrate the expressive power of this construct, consider the following example. Let I be an integer vector variable, and let *Halt* be a boolean vector expression defined as follows: $\sigma(Halt)|_u = false$ if Turing Machine number $\sigma(I)|_u$ stops within $|u|$ steps. Let n be a positive integer. Consider the following program

$$I{:=}n;\ B{:=}Halt;\ Div{:=}true;$$
$$\mathsf{loop}\ \neg B\ \mathsf{do}\ (Div{:=}false;\ B{:=}true)$$

This program always terminates, and $Div|_0$ is true iff Turing Machine number n diverges. Observe that this somewhat surprising fact does not prohibit the existence of some complete proof system, as the Consequence Rule considers all valid formula of the underlying logic as axioms, even though no complete proof system may exist for it.

More conceptually, programs of $\mathcal{L}$ encapsulate two kinds of divergences. The first kind comes from the *virtualisation loops* implicitely specified by each data-parallel assignment to infinite vectors. It may be called *spatial divergence*, and it is not visible at the level of semantics of $\mathcal{L}$. The second kind is due to explicit iterations loops. It may be called *temporal divergence*. It is denoted by $\perp$ in the semantics. Studying the semantics of $\mathcal{L}$ and the completeness of the associated proof systems leads thus to stratify the diverging behaviors of usual scalar programs into these two classes. As such, it definitely deserves further studies.

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 and D. Cachera. On the completeness of a proof system for a simple data-parallel programming language. Research Report 94–42, LIP ENS Lyon, France, December 1994. Available at URL `ftp://ftp.lip.ens-lyon.fr/pub/Rapports/RR/RR94/RR94-42.ps.Z`.
3. L. Bougé, Y. Le Guyadec, G. Utard, and B. Virot. On the expressivity of a weakest preconditions calculus for a simple data-parallel programming language. In *ConPar'94–VAPP VI*, Linz, Austria, September 1994.
4. 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 Applications in Parallel and Distributed Computing*, Caracas, Venezuela, April 1994. IFIP WG 10.3, North-Holland.
5. L. Bougé and J.-L. Levaire. Control structures for data-parallel SIMD languages : semantics and implementation. *Future Generation Computer Systems*, 8:363–378, 1992.
6. 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.
7. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
8. J. Gabarró and R. Gavaldà. An approach to correctness of data parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):185–201, August 1994.
9. M.J.C. Gordon. *Programming Language Theory and its Implementation.* Prentice Hall International, 1988.
10. C.A.R. Hoare. An axiomatic basis for computer programming. *Comm of the ACM*, 12:576–580, 1969.
11. A. Stewart. An axiomatic treatment of SIMD assignment. *BIT*, 30:70–82, 1990.