Proving Data-Parallel Programs : a Unifying Approach

David Cachera, Gil Utard

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Proving data-parallel programs: a unifying approach

David Cachera, Gil Utard

N ° 3032

Novembre 1996

THÈME 1

ISSN 0249-6399

INRIA
RHÔNE-ALPES

Proving data-parallel programs: a unifying approach

David Cachera*, Gil Utard*

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n° 3032 — Novembre 1996 — 15 pages

Abstract: We define an axiomatic semantics for a common kernel of existing data-parallel languages. We introduce an assertional language which enables us to define a weakest liberal precondition calculus which has the Definability Property, and a proof system (*à la* Hoare) which has the Completeness Property. Moreover, our axiomatic semantics integrates two previous works in the definition of proof systems for data-parallel programs. This work sheds a new light on the logical complexity of proving data-parallel programs.

Key-words: Verifying and reasoning about programs, data-parallel languages, proof system, Hoare logic.

(Résumé : tsvp)

This work has been partly supported by the French CNRS Coordinated Research Program on Parallelism, Networks and Systems PRS, and the DRET contract 91/1180.

* Contact: {David.Cachera,Gil.Utard}@ens-lyon.fr. LIP, ENS Lyon, 46 Allée d'Italie, F-69364 Lyon cedex 07, France.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : (33) 76 61 52 00 – Télécopie : (33) 76 61 52 52

Une approche unificatrice dans la preuve de programmes data-parallèles

Résumé : Nous définissons une sémantique axiomatique pour un noyau commun à des langages data-parallèles existants. Nous introduisons un langage d'assertions qui nous permet à la fois de définir un calcul des plus faibles préconditions qui a la propriété de définissabilité, et un système de preuve à la Hoare qui est complet. De plus, notre sémantique axiomatique intègre deux travaux précédents sur la définition de systèmes de preuve pour les programmes data-parallèles. Ce travail apporte un nouveau point de vue sur la complexité de la preuve de programmes data-parallèles.

Mots-clé : Spécification et validation de programmes, langages data-parallèles, système de preuve, logique de Hoare.

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 [6] and denotational semantics [5], and to define a proof system for this language, in the style of the usual Hoare's logic approach [9, 5]. The originality of our approach lies in the treatment of the *extent of parallelism*, that is, the subset of currently active indices at which a vector instruction is to be applied. Previous approaches led to manipulate lists of indices explicitly [7, 12], or to consider context expressions as assertions modifiers [8]. In contrast, our proof system for $\mathcal{L}$ described the activity context by a vector boolean expression distinct from the usual predicates on program variables.

When defining a proof system, two crucial issues have to be established. First, we have to prove that the proof system is *sound*, that is, any provable property of a program is actually valid. This property is usually easy to check. Conversely, we also want the proof system to be *complete*. In other words: *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 main tool usually used to prove completeness is a weakest precondition calculus. In addition, the weakest precondition of any program has to be definable, i.e. there must exist a logical formula denoting it. Nevertheless, we have shown in [4] that our first assertion language isn't expressive enough to ensure the Definability Property. In order to obtain a complete proof system, two ways have been proposed. The first one [3] keeps the same assertion language and proves completeness in a quite intricate manner. The second one [10] redefines the denotational semantics and the assertion language to ensure definability.

We propose here a novel approach, where we introduce another two-part assertion language close to the initial one. The resulting weakest precondition calculus turns out to have the Definability Property. This enables us to define a new sound and complete proof system.

We first present the $\mathcal{L}$ language, and give its denotational semantics. We describe the new assertion language. We then present the proof system. We define a weakest precondition calculus and prove that it has the Definability Property. At this point, we establish the completeness of our proof system. Finally, we relate this approach to [3, 10].

2 The $\mathcal{L}$ Language

An extensive presentation of the $\mathcal{L}$ language can be found in [6]. For the sake of completeness, we briefly recall its denotational semantics as described in [4].

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 $X|_u$. A *vector expression* E can be of the following forms.

```
Y := 0;
loop (Y <> X) do
        where (Y <> X) do
                Y := X;
                X := Min(X|_{This-1}, X|_{This+1}) + 1;
        end
end
```

0	1	0	1	1	1	1	1	0	1	1	1	1	0

0	**1**	0	**1**	**1**	**1**	**1**	**1**	0	**1**	**1**	**1**	**1**	0
0	1	0	1	**2**	**2**	**2**	1	0	1	**2**	**2**	1	0
0	1	0	1	2	**3**	2	1	0	1	2	2	1	0

0	1	0	1	2	3	2	1	0	1	2	2	1	0

Figure 1: For each pixel in image X, this $\mathcal{L}$ program computes its distance to background. The result is stored in X. We give an instance of execution. Bold numbers correspond to active components in the `where` construct.

- A vector variable X.
- A vector constant of integer or boolean type. Constant 1 denotes the vector whose all components have value 1, $True$ and $False$ denote the vectors whose all components are respectively true and false. Constant expression *This* denotes the vector whose value at index u is u: this is the iproc of MPL, the . operator of C*.
- The componentwise combination of vector expressions: for instance, $X + Y$. All usual scalar operators are overloaded with their vector extension.
- It will prove useful to define an additional type of vector expressions: *conditional vector expressions*. $C?E : F$ denotes the vector whose component at index u is $E|_u$ if boolean vector expression C is true at index u, and $F|_u$ otherwise.
- A *fetch* expression: $E|_A$. Consider a fixed index u. Vector expression A is evaluated, then vector expression E, then the result is rearranged so that the value at index u is fetched at the index which is the value of A at u: $(E|_A)|_u = E|_{(A|_u)}$. In MPL, this is denoted router[A].E. In C* and HyperC, this is denoted [A]E.

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 expression E. Observe that E may be a fetch expression, in which case we obtain a get communication: get E from A into Y is the same as $X := E|_A$. Observe we cannot express send communication in this model.

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 boolean expression B evaluates to false at each currently active index. Observe that the activity context is not modified on executing the body, in contrast with the parallel while of MPL and the whilesomewhere of C*. These constructs can be expressed in $\mathcal{L}$ by a where nested in a loop. Our form is therefore more general.

Conditioning: where B do S. The active indices where 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.

The $\mathcal{L}$ language is quite simple, but it is sufficient to express usual data-parallel algorithms. Consider for instance the program displayed on Figure 1. It is a transposition in one dimension of the computation of the distance in a binary image. Image pixels have value 1 and background pixels have value 0. For each image pixel of an image X, the program computes the number of pixels to the nearest background pixel. At each iteration, the previous value of vector X is temporarily stored in vector Y. Each active pixel computes a new distance value from its two neighbors. When a pixel has computed its distance value, it becomes inactive in the next iterations.

2.2 Denotational Semantics of $\mathcal{L}$

We recall the semantics of $\mathcal{L}$ defined in [4] 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. Here are the most interesting rules:

- $\sigma(This)|_u = u$;
- $\sigma(E + F)|_u = \sigma(E)|_u + \sigma(F)|_u$;
- $\sigma(E|_A)|_u = \sigma(E)|_{\sigma(A)|_u}$;
- $\sigma(C?E : F)|_u = \begin{cases} \sigma(E)|_u & \text{if } \sigma(C)|_u \text{ is true} \\ \sigma(F)|_u & \text{otherwise} \end{cases}$

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.

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 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 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 Specification of $\mathcal{L}$ programs

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.

3.1 Assertion language

A crucial property of axiomatic semantics in the usual sequential 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 $\langle W, \mathcal{C}\rangle$, where W is a predicate on vector program variables, and $\mathcal{C}$ is a parallel variable which evaluates into the current activity context.

The structure of predicates on vector variables has to be made precise here. We consider only integer and boolean types.

- A *scalar expression* is a scalar variable ($x, y, u, v, \ldots$), a scalar constant (0, 1, $true$, $false$, etc.), a combination of scalar expressions with some scalar operator, or a vector expression of the programming language subscripted by an scalar expression $E|_u$.
- A *formula* is a scalar expression of boolean type, the combination of formulas with logical operators, or a formula quantified on a scalar variable.
- A (vector) predicate is a formula which is closed with respect to all index and scalar variables. External universal quantification is implicit.

For instance, the following are vector predicates:

$\forall u : X\|_u = 0$	All components of X have value 0
$\forall u : \forall v : X\|_u = X\|_v$	All components of X are equal
$\forall u : X\|_u = Y\|_u$	X and Y have the same value at each index
$\forall u : even(u) \Rightarrow (X\|_{u+1} = X\|_u)$	At all even indices, the right component is the same as the local one
$\exists x : \forall u : X\|_u = x$	X is a constant vector

Observe there is no quantification on vector variables in vector predicates. Observe also that $X = Y$ is *not* a predicate, but a boolean vector expression. The equality predicate is $\forall u : X|_u = Y|_u$.

Since a vector predicate is a formula closed with respect to index and scalar variables, we can define its truth value with respect to an environment in the usual way. If predicate P is true in environment σ, then we write $\sigma \models P$. We are now in position to define the validity of an assertion in a program state.

Definition 1 (Satisfiability) *Let (σ, c) be a state, $\langle W, \mathcal{C}\rangle$ an assertion. We say that state (σ, c) satisfies assertion $\langle W, \mathcal{C}\rangle$, and write $(\sigma, c) \models \langle W, \mathcal{C}\rangle$, if $\sigma[\mathcal{C} \leftarrow c] \models W$. By convention, $\perp$ satisfies any assertion. The set of states satisfying $\langle W, \mathcal{C}\rangle$ is denoted by $[\![\langle W, \mathcal{C}\rangle]\!]$, or sometimes $\langle W, \mathcal{C}\rangle$ when no confusion may arise.*

> A basic instance, $\langle \forall u : X|_u = Y|_u + 1 \wedge \mathcal{C}|_u = (Y|_u = 2), \mathcal{C}\rangle$ is intended to denote (apart $\perp$) the set of states (σ, c) such that: at each index u, the local value $x = \sigma(X)|_u$ of the vector variable X is the local value of the vector variable Y plus 1; the current extent of parallelism c ranges over those indices u such that $\sigma(Y)|_u = 2$ holds.

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 key result is that the usual substitution lemma [1] extends to this new setting.

Lemma 1 *Let P be a predicate on vector variables, X a vector variable, and E a vector expression.*

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

Proof *This is easily proved by induction on the structure of vector predicates and vector expressions. The crucial point is that we only consider here the substitution of a* whole vector X, *in contrast with [1] where the substitution of a particular component $X[u]$ is supported.* ❑

3.2 $\mathcal{C}$-conversion

In the definition of satisfiability, variable $\mathcal{C}$ enabled us to describe context properties in predicate W. The initial value of variable $\mathcal{C}$ is of no significance, since it is overloaded in $\sigma[\mathcal{C} \leftarrow c]$. We introduce a $\mathcal{C}$-conversion mechanism which enables us to rename this context variable. This mechanism is similar to the α-conversion of the lambda calculus. We follow the notation of [1], and denote by $Var(S)$ the set of variables appearing in program S. Similarly, let $Var(W)$ be the set of variables which appear in predicate W. The value of W depends on these variables only.

Proposition 1 ($\mathcal{C}$-conversion) *Let W be a predicate and $\mathcal{C}'$ such that $\mathcal{C}' \notin Var(W)$. We have*

$$[\![\langle W, \mathcal{C}\rangle]\!] = [\![\langle W[\mathcal{C}'/\mathcal{C}], \mathcal{C}'\rangle]\!] .$$

Proof *Let $(\sigma, c) \in [\![\langle W, \mathcal{C}\rangle]\!]$. By definition, $\sigma[\mathcal{C} \leftarrow c] \models W$. As $\mathcal{C}' \notin Var(W)$, $\sigma[\mathcal{C} \leftarrow c][\mathcal{C}' \leftarrow c] \models W$. Let us denote $\sigma[\mathcal{C}' \leftarrow c]$ by σ' We thus have $\sigma'[\mathcal{C} \leftarrow \sigma'(\mathcal{C}')] \models W$. By the Substitution Lemma, we conclude that $\sigma[\mathcal{C}' \leftarrow c] \models W[\mathcal{C}'/\mathcal{C}]$, that is $(\sigma, c) \in [\![\langle W[\mathcal{C}'/\mathcal{C}], \mathcal{C}'\rangle]\!]$. Conversely, let $(\sigma, c) \in [\![\langle W[\mathcal{C}'/\mathcal{C}], \mathcal{C}'\rangle]\!]$. We have $\sigma[\mathcal{C}' \leftarrow c] \models W[\mathcal{C}'/\mathcal{C}]$. By the Substitution Lemma, we have $\sigma'[\mathcal{C} \leftarrow \sigma'(\mathcal{C}')] \models W$, so $\sigma[\mathcal{C} \leftarrow c][\mathcal{C}' \leftarrow c] \models W$. As $\mathcal{C}' \notin Var(W)$, we conclude that $\sigma[\mathcal{C} \leftarrow c] \models W$.* ❑

3.3 Assertion Implication

The $\mathcal{C}$-conversion mechanism gives an equivalence property between assertions containing the same predicate *modulo* a context substitution. But we need a weaker and more general relation between assertions, namely assertion implication.

Definition 2 (Assertion Implication) *Let $\langle W, \mathcal{C}\rangle$ and $\langle W', \mathcal{C}'\rangle$ be two assertions. We say that $\langle W, \mathcal{C}\rangle$ implies $\langle W', \mathcal{C}'\rangle$, and we write $\langle W, \mathcal{C}\rangle \Rightarrow \langle W', \mathcal{C}'\rangle$, in the two following cases:*

- *$\mathcal{C}$ is the same variable as $\mathcal{C}'$ and $W \Rightarrow W'$ in the usual (first-order logic) sense;*
- *$\mathcal{C}$ is distinct from $\mathcal{C}'$ and $W[\mathcal{C}''/\mathcal{C}] \Rightarrow W'[\mathcal{C}''/\mathcal{C}']$, where $\mathcal{C}'' \notin Var(W) \cup Var(W')$.*

Note that the second case corresponds to a simultaneous $\mathcal{C}$-conversion of both assertions, and that $\mathcal{C}''$ is a new variable that doesn't appear neither in W nor in W'. Finding such a $\mathcal{C}''$ is always possible, since all expressions in our language are finite terms. Note also that, if $\mathcal{C}$ doesn't appear in W'' (resp. $\mathcal{C}'$ in W), chosing $\mathcal{C}'' = \mathcal{C}$ (resp. $\mathcal{C}'' = \mathcal{C}'$) is correct and requires only one substitution.

We now have to show that this definition of assertion implication respects the semantics given to assertions.

Proposition 2 (Assertion Implication) *Let $\langle W, \mathcal{C}\rangle$ and $\langle W', \mathcal{C}'\rangle$ be two assertions. Then*

$$\langle W, \mathcal{C}\rangle \Rightarrow \langle W', \mathcal{C}'\rangle \qquad \Leftrightarrow \qquad [\![\langle W, \mathcal{C}\rangle]\!] \subseteq [\![\langle W', \mathcal{C}'\rangle]\!]$$

Proof *We consider two cases:*

- *$\mathcal{C}$ and $\mathcal{C}'$ are the same variable. Let us assume that $\langle W, \mathcal{C}\rangle \Rightarrow \langle W', \mathcal{C}\rangle$. By definition, it simply means that $W \Rightarrow W'$. Let (σ, c) be in $[\![\langle W, \mathcal{C}\rangle]\!]$. We have $\sigma[\mathcal{C} \leftarrow c] \models W$, so $\sigma[\mathcal{C} \leftarrow c] \models W'$: $(\sigma, c) \in [\![\langle W', \mathcal{C}\rangle]\!]$. The converse is also trivial.*
- *$\mathcal{C}$ is distinct from $\mathcal{C}'$. Let $\mathcal{C}''$ be a variable such that $\mathcal{C}'' \notin Var(W) \cup Var(W')$. By the $\mathcal{C}$-conversion Property, $[\![\langle W, \mathcal{C}\rangle]\!] = [\![\langle W_1, \mathcal{C}''\rangle]\!]$ and $[\![\langle W', \mathcal{C}'\rangle]\!] = [\![\langle W'_1, \mathcal{C}''\rangle]\!]$, where $W_1 = W[\mathcal{C}''/\mathcal{C}]$ and $W'_1 = W'[\mathcal{C}''/\mathcal{C}']$. We just go back to the first case: $[\![\langle W_1, \mathcal{C}''\rangle]\!] \subseteq [\![\langle W'_1, \mathcal{C}''\rangle]\!]$ if and only if $W_1 \Rightarrow W'_1$.*

❑

3.4 Specification Validity

We can define the validity of a specification of a $\mathcal{L}$ program with respect to its denotational semantics. Because $\perp$ satisfies any assertion, our definition of validity is relative to partial correctness.

Definition 3 (Specification Validity) *Let S be a $\mathcal{L}$ program, $\langle V, \mathcal{C}\rangle$ and $\langle W, \mathcal{C}'\rangle$ two assertions. We say that specification $\langle V, \mathcal{C}\rangle$ S $\langle W, \mathcal{C}'\rangle$ is valid, denoted by $\models \langle V, \mathcal{C}\rangle$ S $\langle W, \mathcal{C}'\rangle$, if for all states (σ, c)*

$$((\sigma, c) \models \langle V, \mathcal{C}\rangle) \Rightarrow ([\![S]\!](\sigma, c) \models \langle W, \mathcal{C}'\rangle).$$

4 Proof System

Thanks to the $\mathcal{C}$-conversion, we describe a proof system below, where we always assume that context variable $\mathcal{C}$ is not in $Var(S)$.

Rule 1 (Assignment) *We extend the usual backwards axiom by taking into consideration that vector variable X is modified only at active indices.*

$$\langle W[\mathcal{C}?E{:}X/X], \mathcal{C}\rangle \ X{:=}E \ \langle W, \mathcal{C}\rangle$$

Rule 2 (Sequencing) *It is a straightforward generalization of the usual case.*

$$\frac{\langle W, \mathcal{C}\rangle \ S \ \langle V, \mathcal{C}\rangle \quad \langle V, \mathcal{C}\rangle \ T \ \langle U, \mathcal{C}\rangle}{\langle W, \mathcal{C}\rangle \ S;T \ \langle U, \mathcal{C}\rangle}$$

Rule 3 (Conditioning) *Following the denotational semantics, the context part is the conjunction of the previous context expression and the condition of the conditioning construct. This is denoted in the substitution of $\mathcal{C}'$ by $\mathcal{C} \wedge B$.*

$$\frac{\langle W, \mathcal{C}'\rangle \ S \ \langle V, \mathcal{C}'\rangle, \mathcal{C}' \notin Var(V)}{\langle W[\mathcal{C} \wedge B/\mathcal{C}'], \mathcal{C}\rangle \ \mathsf{where}\ B\ \mathsf{do}\ S \ \langle V, \mathcal{C}\rangle}$$

Rule 4 (Iteration) *The usual loop invariant here has to be invariant with respect to both the variable values and the activity context.*

$$\frac{\langle I \wedge \exists u : (\mathcal{C}|_u \Rightarrow B|_u), \mathcal{C}\rangle \ S \ \langle I, \mathcal{C}\rangle}{\langle I, \mathcal{C}\rangle \ \mathsf{while}\ B\ \mathsf{do}\ S \ \langle I \wedge \forall u : (\mathcal{C}|_u \Rightarrow \neg B|_u), \mathcal{C}\rangle}$$

Rule 5 (Consequence) *Following Definition 2, we can state the consequence rule.*

$$\frac{\langle W', \mathcal{C}'\rangle \Rightarrow \langle W, \mathcal{C}\rangle \quad \langle W, \mathcal{C}\rangle \ S \ \langle V, \mathcal{C}\rangle \quad \langle V, \mathcal{C}\rangle \Rightarrow \langle V', \mathcal{C}''\rangle}{\langle W', \mathcal{C}'\rangle \ S \ \langle V', \mathcal{C}'\rangle}$$

If a specification formula $\langle V, \mathcal{C}\rangle \ S \ \langle W, \mathcal{C}'\rangle$ is derivable in the proof system, then we write $\vdash \langle V, \mathcal{C}\rangle \ S \ \langle W, \mathcal{C}'\rangle$.

Theorem 1 (Soundness of $\vdash$) *The $\vdash$ proof system is sound: If*

$$\vdash \langle V, \mathcal{C}\rangle \ S \ \langle W, \mathcal{C}'\rangle$$

then

$$\models \langle V, \mathcal{C}\rangle \ S \ \langle W, \mathcal{C}'\rangle.$$

Proof *The proof is straightforward. As usual, we just have to prove soundness for each rule of the proof system.* ❑

Example

Let us prove the correctness of the small program given on Figure 1, page 4. At the beginning, there is a binary image in X and all indices are active. On termination, each component $X|_u$ holds the distance to the background. This distance is denoted by $D|_u$.

To find an invariant, observe Figure 2. We express that components $X|_u$ such that $X|_u = Y|_u$ have already computed their distance. Moreover, it is also true for their left and right neighbors. Otherwise, components $X|_u$ such that $X|_u \neq Y|_u$ have value k (which is the number of iterations.) Moreover, left and right neighbors which have already computed their distance (i.e. such that $X|_{u\pm1} = Y|_{u\pm1}$) are equal to $k-1$. More formally, the invariant is

$$I \equiv \exists k : \forall u : C|_u = true \wedge \begin{cases} X|_u = Y|_u \ \Rightarrow \begin{cases} X|_u = D|_u \\ X|_{u-1} = D|_{u-1} \\ X|_{u+1} = D|_{u+1} \end{cases} \\ X|_u \neq Y|_u \ \Rightarrow \begin{cases} X|_u = k \wedge k \leq D|_u \\ X|_{u-1} = Y|_{u-1} \ \Rightarrow \ X|_{u-1} = k-1 \\ X|_{u+1} = Y|_{u+1} \ \Rightarrow \ X|_{u+1} = k-1 \end{cases} \end{cases}$$

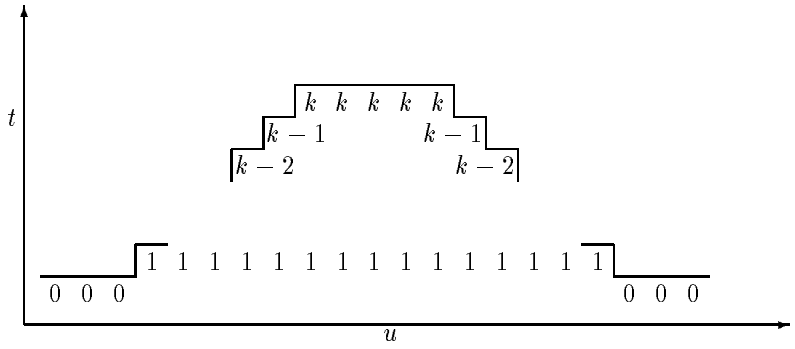

Figure 2: Evolution of X. At iteration k, component $X|_u$ such that distance to background is greater or equal to k are active. Other components are inactive, and their value is equal to the distance. It is also true for their left and right neighbors.

Following Iteration Rule 4, if the invariant is valid, then the final wanted postcondition holds:

$$\langle I \wedge \forall u : X|_u = Y|_u, \mathcal{C} \rangle \Rightarrow \langle \forall u : X|_u = D|_u \wedge \mathcal{C}|_u = true, \mathcal{C} \rangle$$

Now, we prove that the invariant is valid. Let S be the body of the loop, i.e. the where construct. We denote the minimum of $X|_{v-1}$ and $X|_{v+1}$ by Min_v. Consider the predicate P described below

$$P \equiv \exists k : \forall u : C|_u = true \wedge$$

$$\begin{cases} X|_u = Y|_u \Rightarrow \\ \quad (1) \begin{cases} X|_u = D|_u \\ X|_{u-1} = Y|_{u-1} \Rightarrow X|_{u-1} = D|_{u-1} \\ X|_{u-1} \neq Y|_{u-1} \Rightarrow Min_{u-1} + 1 = D|_{u-1} \\ X|_{u+1} = Y|_{u+1} \Rightarrow X|_{u+1} = D|_{u+1} \\ X|_{u+1} \neq Y|_{u+1} \Rightarrow Min_{u+1} + 1 = D|_{u+1} \end{cases} \\ \\ X|_u \neq Y|_u \Rightarrow \\ \quad \begin{cases} Min_u + 1 = X|_u \Rightarrow \\ \quad (2) \begin{cases} Min_u + 1 = D|_u \\ X|_{u-1} = Y|_{u-1} \Rightarrow X|_{u-1} = D|_{u-1} \\ X|_{u-1} \neq Y|_{u-1} \Rightarrow Min_{u-1} + 1 = D|_{u-1} \\ X|_{u+1} = Y|_{u+1} \Rightarrow X|_{u+1} = D|_{u+1} \\ X|_{u+1} \neq Y|_{u+1} \Rightarrow Min_{u+1} + 1 = D|_{u+1} \end{cases} \\ \\ Min_u + 1 \neq X|_u \Rightarrow \\ \quad (3) \begin{cases} Min_u + 1 = k \wedge k \leq D|_u \\ X|_{u-1} = Y|_{u-1} \Rightarrow X|_{u-1} = k - 1 \\ X|_{u-1} \neq Y|_{u-1} \Rightarrow Min_{u-1} + 1 = X|_{u-1} \Rightarrow Min_{u-1} + 1 = k - 1 \\ X|_{u+1} = Y|_{u+1} \Rightarrow X|_{u+1} = k - 1 \\ X|_{u+1} \neq Y|_{u+1} \Rightarrow Min_{u+1} + 1 = X|_{u+1} \Rightarrow Min_{u+1} + 1 = k - 1 \end{cases} \end{cases} \end{cases}$$

It is easy, but tedious, to show that $\langle P, \mathcal{C} \rangle\ S\ \langle I, \mathcal{C} \rangle$.

Following Iteration Rule 4 and Consequence Rule 5, we must prove that $I \Rightarrow P$. Let k be such that I is true. We show that P holds for $k + 1$. Observe that the definition of distance has the following property: $\forall u : D|_u = Min(D|_{u-1}, D|_{u+1}) + 1$.

Proof

(1) $X|_u = Y|_u$. *By hypothesis* $X|_u = D|_u$, $X|_{u-1} = D|_{u-1}$, *and* $X|_{u+1} = D|_{u+1}$. *Let us focus on* $X|_{u-1}$. *If* $X|_{u-1} = Y|_u$ *then* $X|_{u-1} = D|_{u-1}$, *otherwise* $X|_{u-1} = k$, $X|_u = k - 1$ *and* $X|_{u-2} \geq k - 1$. *Therefore,* $D|_{u-1} = Min(X|_{u-2}, X|_u) + 1 = k$. *The proof is similar for* $X|_{u+1}$.

(2) $X|_u \neq Y|_u$ *and* $Min(X|_{u-1}, X|_{u+1}) = k - 1$. *By hypothesis,* $X|_u = k$ *and there are two cases to be considered. In the first case,* $X|_{u-1} = k-1$ *and* $X|_{u+1} \geq k-1$. *The second case is symmetrical. Let us*

focus on the first one. As $X|_{u-1} \neq k$, we deduce ad absurdum $X|_{u-1} = Y|_{u-1}$. Thus, $X|_{u-1} = D|_{u-1}$ and $X|_u = D|_u = Min(X|_{u-1}, X|_{u+1}) + 1$. Let us now consider $X|_{u+1}$. If $X|_{u+1} = Y|_{u+1}$, then $X|_{u+1} = k - 1 = D|_{u+1}$; otherwise, $X|_{u+1} = k$ and there are two cases to be considered.

$X|_{u+2} = Y|_{u+2}$. Consequently $X|_{u+2} = k - 1 = D|_{u+2}$ and $X|_{u+1} = k = D|_{u+1}$. Therefore, $Min(X|_u, X|_{u+2}) + 1 = k = D|_{u+1}$.

$X|_{u+2} \neq Y|_{u+2}$. Consequently $X|_{u+2} = k \leq D|_{u+2}$ and $Min(X|_u, X|_{u+2}) = k$. As $X|_u = D|_u = k$, we have $Min(D|_u, D|_{u+2}) = k$. Therefore, $D|_{u+1} = Min(D|_u, D|_{u+1}) + 1 = Min(X|_u, X|_{u+1}) + 1$.

(3) $X|_u \neq Y|_u$. By hypothesis $X|_u = k$ and $Min(X|_{u-1}, X|_{u+1}) \neq k-1$, we deduce a.a. *$X|_{u-1} \neq Y|_{u-1}$ and $X|_{u+1} \neq Y|_{u+1}$, i.e. $X|_{u-1} = k$ and $X|_{u+1} = k$. Therefore, that $Min(X|_{u-1}, X|_{u+1}) + 1 = k + 1$. As $Min(D|_{u-1}, D|_{u+1}) \geq k$, we deduce $D|_u \geq k+1$. Let us focus on $X|_{u-1}$. If $Min(X|_{u-2}, X|_u) + 1 = X|_{u-1}$, then $Min(X|_{u-2}, X|_u) + 1 = k$. The proof is similar for $X|_{u+1}$.*

□

We have proved that $\langle I \wedge \exists u : X|_u \neq Y|_u, \mathcal{C}\rangle \Rightarrow \langle P, \mathcal{C}\rangle$. Every binary image X such that $\forall u : X|_u \in \{0, 1\}$ validates our invariant I. Observe that if X is such that $\forall u : X|_u = 1$, then our program does not terminate. This is also valid, since our proof system only deals with *partial correctness*.

5 Weakest Preconditions Calculus and Completeness

We now want to address the *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.

5.1 Weakest Preconditions Calculus

Our main tool to prove the completeness of our system is a weakest preconditions calculus. Let us first motivate its use. We want to prove that

$$\models \langle V, \mathcal{C}'\rangle \ S \ \langle W, \mathcal{C}\rangle \quad \Rightarrow \quad \vdash \langle V, \mathcal{C}'\rangle \ S \ \langle W, \mathcal{C}\rangle.$$

Let $\langle V, \mathcal{C}'\rangle \ S \ \langle W, \mathcal{C}\rangle$ be a valid specification formula. Assume for a while that we can find an assertion $\langle V', \mathcal{C}''\rangle$ such that $\vdash \langle V', \mathcal{C}''\rangle \ S \ \langle W, \mathcal{C}\rangle$ holds, and moreover that $\langle V, \mathcal{C}'\rangle \Rightarrow \langle V', \mathcal{C}''\rangle$. Then, using the Consequence Rule, we have proved $\vdash \langle V, \mathcal{C}'\rangle \ S \ \langle W, \mathcal{C}\rangle$.

As we only prove (partial) correctness and not termination, we use a weakest *liberal* preconditions calculus.

Definition 4 (Weakest Liberal Preconditions) *Let S be a $\mathcal{L}$ program and $\mathcal{E}$ a subset of* State.

$$wlp(S, \mathcal{E}) = \{(\sigma, c) \mid [\![S]\!](\sigma, c) \in \mathcal{E}\}$$

As a convention, we will denote by $wlp(\mathrm{S}, \langle W, \mathcal{C}\rangle)$ the weakest precondition of program S and subset $[\![\langle W, \mathcal{C}\rangle]\!]$.

Lemma 2 (Consequence Lemma) $\models \langle V, \mathcal{C}'\rangle \ S \ \langle W, \mathcal{C}\rangle$ *iff* $[\![\langle V, \mathcal{C}'\rangle]\!] \subseteq wlp(S, \langle W, \mathcal{C}\rangle)$.

Proof *Let us assume that $\models \langle V, \mathcal{C}'\rangle \ S \ \langle W, \mathcal{C}\rangle$, and let (σ, c) be in $[\![\langle V, \mathcal{C}'\rangle]\!]$. By definition of specification validity, we have $[\![S]\!](\sigma, c) \models \langle W, \mathcal{C}\rangle$, i.e. $(\sigma, c) \in wlp(S, \langle W, \mathcal{C}\rangle)$ by definition of weakest liberal preconditions. Conversely, let us assume that $[\![\langle V, \mathcal{C}'\rangle]\!] \subseteq wlp(S, \langle W, \mathcal{C}\rangle)$, and let s be a state such that $s \models \langle V, \mathcal{C}'\rangle$. If s is $\perp$, then $[\![S]\!](s) = \perp \models \langle W, \mathcal{C}\rangle$. If $s = (\sigma, c)$, since we know by hypothesis that $s \in wlp(S, \langle W, \mathcal{C}\rangle)$, we have $[\![S]\!](\sigma, c) \models \langle W, \mathcal{C}\rangle$.* □

The weakest preconditions defined above are sets of states. As such, they cannot be explicitly manipulated in the proof system. We have to prove that these particular sets of states can actually be described by suitable assertions. This is the *definability* problem. The particular form of our assertions helps us to establish the following result.

Proposition 3 (Definability of wlp) *Let S be a $\mathcal{L}$ program, and $\langle W, \mathcal{C}\rangle$ an assertion. There exists an assertion $\langle V, \mathcal{C}'\rangle$ such that*

$$[\![\langle V, \mathcal{C}'\rangle]\!] = wlp(S, [\![\langle W, \mathcal{C}\rangle]\!]).$$

The proof of this property relies on a number of lemmas detailled below. The general method for proving Lemmas 3 and 4 is the same: we first define an assertion that is constructed from $\langle V, \mathcal{C}'\rangle$. This step is closely related to the rules defined for the proof system. We then prove that this assertion is equal to $wlp(S, [\![\langle W, \mathcal{C}\rangle]\!])$. The only case where we are not able to construct the assertion directly from $\langle V, \mathcal{C}'\rangle$ is the case of loops. In that case, a more complex encoding scheme is needed.

Lemma 3 (Assignment) *Let X be a variable, E an expression, and $\langle W, \mathcal{C}\rangle$ an assertion. Then there exists an assertion $\langle V, \mathcal{C}'\rangle$ such that*

$$wlp(X{:=}E, \langle W, \mathcal{C}\rangle) = [\![\langle V, \mathcal{C}'\rangle]\!] .$$

Proof *In the case where $X = \mathcal{C}$, let us take a new variable $\mathcal{C}'$ distinct from X, and define $\langle W', \mathcal{C}'\rangle$ as a $\mathcal{C}$-conversion of $\langle W, \mathcal{C}\rangle$. Otherwise take $W' = W$ and $\mathcal{C}' = \mathcal{C}$. Take now $V = W'[(\mathcal{C}'?E{:}X)/X]$. As assertions $\langle W, \mathcal{C}\rangle$ and $\langle W', \mathcal{C}'\rangle$ are equivalent, $wlp(X{:=}E, \langle W, \mathcal{C}\rangle) = wlp(X{:=}E, \langle W', \mathcal{C}'\rangle)$.*

Let now s be in $wlp(X{:=}E, \langle W', \mathcal{C}'\rangle)$. If $s = \bot$, then $s \in [\![\langle V, \mathcal{C}'\rangle]\!]$. Otherwise, $s = (\sigma, c)$ for some environment σ and some context c. Let (σ', c) be $[\![S]\!](\sigma, c)$. By definition of s we have $\sigma'[\mathcal{C}' \leftarrow c] \models W'$. But σ' is $\sigma[X \leftarrow c?\sigma(E){:}\sigma(X)]$. We thus have $\sigma[\mathcal{C}' \leftarrow c][X \leftarrow \sigma[\mathcal{C}' \leftarrow c](\mathcal{C}'?E{:}X)] \models W'$. By the Substitution Lemma, we conclude that $\sigma[\mathcal{C}' \leftarrow c] \models V$.

Conversely, let s be in $[\![\langle V, \mathcal{C}'\rangle]\!]$. If $s = \bot$, then $s \in wlp(X{:=}E, \langle W', \mathcal{C}'\rangle)$. Otherwise, $s = (\sigma, c)$ for some environment σ and some context c. Let (σ', c) be $[\![S]\!](\sigma, c)$: σ' is $\sigma[X \leftarrow c?\sigma(E){:}\sigma(X)]$. We have $\sigma'[\mathcal{C}' \leftarrow c] = \sigma[X \leftarrow c?\sigma(E){:}\sigma(X)][\mathcal{C}' \leftarrow c] = \sigma[\mathcal{C}' \leftarrow c][X \leftarrow \sigma[\mathcal{C}' \leftarrow c](\mathcal{C}'?E{:}X)]$. By the Substitution Rule, we thus have

$$\sigma'[\mathcal{C}' \leftarrow c] \models W'.$$

❑

Lemma 4 (Conditioning) *Let S be a program, B a boolean vector expression, $\langle W, \mathcal{C}\rangle$ an assertion such that $\mathcal{C} \notin Var(B) \cup Change(S)$, and $\mathcal{C}'$ a variable such that $\mathcal{C}' \notin Var(W)$. If there exists a predicate W' such that*

$$wlp(S, \langle W, \mathcal{C}'\rangle) = [\![\langle W', \mathcal{C}'\rangle]\!],$$

then

$$wlp(\mathsf{where}\ B\ \mathsf{do}\ S, \langle W, \mathcal{C}\rangle) = [\![\langle W'[\mathcal{C} \wedge B/\mathcal{C}'], \mathcal{C}\rangle]\!] .$$

Proof *As in the previous lemmas, the case of undefined states ($\bot$) is trivial.*

Let us first prove that

$$wlp(\mathsf{where}\ B\ \mathsf{do}\ S, \langle W, \mathcal{C}\rangle) \subseteq [\![\langle W'[\mathcal{C} \wedge B/\mathcal{C}'], \mathcal{C}\rangle]\!] .$$

Let $(\sigma, c) \in wlp(\mathsf{where}\ B\ \mathsf{do}\ S, \langle W, \mathcal{C}\rangle)$. We have $[\![\mathsf{where}\ B\ \mathsf{do}\ S]\!](\sigma, c) = (\sigma', c)$, where $(\sigma', c \wedge \sigma(B)) = [\![S]\!](\sigma, c \wedge \sigma(B))$. By definition, $(\sigma', c) \models \langle W, \mathcal{C}\rangle$. Let us denote $\sigma'[\mathcal{C} \leftarrow c]$ by σ'_c, and $\sigma[\mathcal{C} \leftarrow c]$ by σ_c. We have $\sigma'_c \models W$. As $\mathcal{C}' \notin Var(W)$, we also have $\sigma'_c[\mathcal{C}' \leftarrow c \wedge \sigma(B)] \models W$, that can be rewritten into $(\sigma'_c, c \wedge \sigma(B)) \models \langle W, \mathcal{C}'\rangle$. We have $\mathcal{C} \notin Change(S)$, so $[\![S]\!](\sigma_c, c \wedge \sigma(B)) = (\sigma'_c, c \wedge \sigma(B))$, and then $(\sigma_c, c \wedge \sigma(B)) \models \langle W', \mathcal{C}'\rangle$. We rewrite it into $\sigma_c[\mathcal{C}' \leftarrow c \wedge \sigma(B)] \models W'$. Since $\mathcal{C} \notin Var(B)$, we have $\sigma_c[\mathcal{C}' \leftarrow \sigma_c(\mathcal{C} \wedge B)] \models W'$, and finally $\sigma_c \models W'[\mathcal{C} \wedge B/\mathcal{C}']$ by the Substitution Lemma.

We conclude that $(\sigma, c) \models \langle W'[\mathcal{C} \wedge B/\mathcal{C}'], \mathcal{C}\rangle$.

The other inclusion is proved exactly the same way.

❑

Lemma 5 (Iteration) *Let T be a $\mathcal{L}$ program, and $\langle W, \mathcal{C}\rangle$ an assertion. There exists an assertion $\langle V, \mathcal{C}'\rangle$ such that*

$$[\![\langle V, \mathcal{C}'\rangle]\!] = wlp(\mathsf{loop}\ B\ \mathsf{do}\ T, [\![\langle W, \mathcal{C}\rangle]\!]).$$

Proof *Let S be the $\mathcal{L}$ program $\mathsf{loop}\ B\ \mathsf{do}\ T$. The proof starts from a meta-mathematical description of wlp. A predicate in the assertion langage is constructed in a way like a Gödel encoding scheme. This is quite a long proof, so we only show the starting point and the sketch of the proof argument. The proof begins by showing that a state (σ, c) is in the weakest liberal precondition of the loop $wlp(\mathsf{loop}\ B\ \mathsf{do}\ T, \langle W, \mathcal{C}\rangle)$ iff $\sigma^c = \sigma[\mathcal{C}' \leftarrow c]$ $(\mathcal{C}' \notin Var(S))$ satisfies the following:*

$$\forall k : \forall \sigma_0, \sigma_1, \ldots, \sigma_k : \begin{cases} \textit{If} & \sigma^c = \sigma_0 \\ \textit{and} & \forall i \in [0, k[: \begin{cases} \sigma_i \models \exists u : B|_u \wedge \mathcal{C}'|_u \\ [\![T]\!]((\sigma_i, \sigma_i(\mathcal{C}'))) = (\sigma_{i+1}, \sigma_{i+1}(\mathcal{C}')) \end{cases} \\ \textit{then} & \sigma_k \models (W \wedge \mathcal{C} = \mathcal{C}') \vee (\exists u : B|_u \wedge \mathcal{C}'|_u) \end{cases}$$

Denote by $\mathcal{V}$ this property. This property is clearly true because it corresponds to the denotational description of iteration. Since each environment σ_i can be denoted by an integer s_i (there is a finite number of finite vectors), the second step shows that $[\![T]\!]((\sigma_i, \sigma_i(\mathcal{C}'))) = (\sigma_{i+1}, \sigma_{i+1}(\mathcal{C}'))$ is equivalent to

$$(\sigma_i, c) \in (wlp(T, \langle \bar{X} = s_{i+1}, \mathcal{C}\rangle) \wedge \neg wlp(T, \langle false, \mathcal{C}\rangle))$$

where $\bar{X}$ denotes all variables that are defined in the environment and $[\![\langle \bar{X} = s_{i+1}, \mathcal{C}\rangle]\!] = \{\sigma_{i+1}\}$. By induction hypothesis, this weakest liberal preconditions has the Definability Property. A predicate V' can be defined such that $\mathcal{V}$ is equivalent to $\forall k : \forall s_0, s_1, \ldots, s_k : V'$. The last step of the proof consists in encoding the sequence of integers $s_0, s_1, \ldots, s_k$ by a Gödel predicate. The property $\mathcal{W}$ is then denoted by $\forall k : \forall s : V''$, which is our final predicate V.

Interested readers can find more details of this proof technique in [2] or an introduction in [14].

❑

Thank to these lemmas, we now can give the proof of Proposition 3.

Proof *The proof is by induction on the structure of S. The case of assignment is directly given by Lemma 3. The case of sequencing is simply treated by observing that $wlp(S;T, \langle W, \mathcal{C}\rangle) = wlp(S, wlp(T, \langle W, \mathcal{C}\rangle))$.*

Let us detail a bit more the case of conditioning. Our aim is to find an assertion defining $wlp(\mathsf{where}\ B\ \mathsf{do}\ S, \langle W, \mathcal{C}\rangle)$. If $\mathcal{C} \in Var(B) \cup Change(S)$, we "pick up" a new $\mathcal{C}''$ such that $\mathcal{C}'' \notin Var(B) \cup Change(S)$ and obtain an assertion $\langle W'', \mathcal{C}''\rangle$ by $\mathcal{C}$-conversion of $\langle W, \mathcal{C}\rangle$. Let us now take a new $\mathcal{C}'$ such that $\mathcal{C}' \notin Var(W) \cup Change(S)$. By induction hypothesis, we are able to find an assertion $\langle W', C'''\rangle$ corresponding to $wlp(S, \langle W'', \mathcal{C}'\rangle)$. But, since $\mathcal{C}' \notin Change(S)$, we have in fact $\mathcal{C}''' = \mathcal{C}'$ (the $\mathcal{C}$-conversion in the case of assignment is never performed). Finally, as $\mathcal{C}'$ and $\mathcal{C}''$ meet the requirements of Lemma 4, we have $[\![\langle W'[\mathcal{C}'' \wedge B/\mathcal{C}'], \mathcal{C}''\rangle]\!] = wlp(\mathsf{where}\ B\ \mathsf{do}\ S, \langle W'', C''\rangle) = wlp(\mathsf{where}\ B\ \mathsf{do}\ S, \langle W, \mathcal{C}\rangle)$.

❑

Lemma 6 (Propagation Lemma) *Let S be a program and $\langle W, \mathcal{C}\rangle$ an assertion. If X is a variable such that $X \notin Var(S) \cup Var(W) \cup \{\mathcal{C}\}$, then there exist an assertion $\langle V, \mathcal{C}'\rangle$ such that $[\![\langle V, \mathcal{C}'\rangle]\!] = wlp(S, [\![\langle W, \mathcal{C}\rangle]\!])$ and $X \notin Var(V) \cup \{\mathcal{C}'\}$.*

Proof *We just give the idea of the proof, which follows exactly the proof of the Definability Property. Let us first see the case of assignment. If X is a variable that is not in $Var(Y{:=}E) \cup Var(W) \cup \{\mathcal{C}\}$, then Lemma 3 gives an assertion $\langle V, \mathcal{C}'\rangle$ such that $[\![\langle V, \mathcal{C}'\rangle]\!] = wlp(Y{:=}E, \langle W, \mathcal{C}\rangle)$ and $X \notin Var(V) \cup \{\mathcal{C}'\}$: when we choose $\mathcal{C}'$, we just have to take a variable distinct from X. The cases of sequencing and conditioning are just propagation cases: since the Propagation Property is true for assignments, is will also be true for compound statements. We just have to assume that, each time we take a "new" context variable ($\mathcal{C}', \mathcal{C}'' \ldots$), we choose it distinct from X. Finally, in the case of iterations, we also have to assume, and it is always possible, that X doesn't appear in the invariant.* ❑

5.2 Completeness

We now want to establish the completeness for our proof system. More formally, we want to prove the following theorem.

Theorem 2 (Completeness) *Let S be a $\mathcal{L}$ program, and $\langle V, \mathcal{C}'\rangle$ and $\langle W, \mathcal{C}\rangle$ two assertions. If $\models \langle V, \mathcal{C}'\rangle\ S\ \langle W, \mathcal{C}\rangle$, then $\vdash \langle V, \mathcal{C}'\rangle\ S\ \langle W, \mathcal{C}\rangle$.*

Proof

The proof of this theorem follows the lines of [1]. It uses the weakest preconditions calculus. For any program S and any assertion $\langle W, \mathcal{C}\rangle$ there exists some assertion $\langle W', \mathcal{C}''\rangle$ such that $[\![\langle W', \mathcal{C}''\rangle]\!] = wlp(S, \langle W, \mathcal{C}\rangle)$. Using the Consequence Rule, it suffices to demonstrate that $\vdash \langle wlp(S, \langle W, \mathcal{C}\rangle)\rangle\ S\ \langle W, \mathcal{C}\rangle$.

The proof is by induction on the structure of S, using the definability properties of the lemmas proved above. The only case we develop here is the case of conditioning: the cases of assignment, sequencing and iteration are simple and classical. Let S be $\mathsf{where}\ B\ \mathsf{do}\ T$. If $\mathcal{C} \in Var(B) \cup Change(S)$, we take a "new"

$\mathcal{C}'''$ such that $\mathcal{C}''' \notin Var(B) \cup Change(S)$, and $\langle W'', \mathcal{C}''' \rangle$ as the $\mathcal{C}$-conversion of $\langle W, \mathcal{C} \rangle$. Let now $\mathcal{C}''$ be a variable such that $\mathcal{C}'' \notin Var(W)$. By the Definability Property, there exists a predicate W' such that

$$wlp(S, \langle W'', \mathcal{C}'' \rangle) = [\![\langle W', \mathcal{C}'' \rangle]\!] .$$

By induction hypothesis, we have $\vdash \langle W', \mathcal{C}'' \rangle \; T \; \langle W'', \mathcal{C}'' \rangle$. Using the Conditioning Rule, we get

$$\vdash \langle W'[\mathcal{C}''' \wedge B/\mathcal{C}''], \mathcal{C}''' \rangle \; S \; \langle W'', \mathcal{C}''' \rangle.$$

As $\mathcal{C} \notin Var(W'')$, Lemma 6 shows that $\mathcal{C} \notin Var(W'')$. Using the Substitution Rule together with the $\mathcal{C}$-conversion mechanism on both ends of the specification thus yields

$$\vdash \langle W'[\mathcal{C} \wedge B/\mathcal{C}''], \mathcal{C} \rangle \; S \; \langle W, \mathcal{C} \rangle,$$

Thanks to Lemma 4, this rewrites into $\vdash wlp(\mathsf{where}\ B\ \mathsf{do}\ T, \langle W, \mathcal{C} \rangle)\ S\ \langle W, \mathcal{C} \rangle$.

---

6 Related Works

The proof system presented here is the result of preliminary works made to establish the completeness of a previous proof system for $\mathcal{L}$ programs [13, 3]. This work has led to the definition of the assertion language used in this paper. The initial proof system uses another assertion language. This language is also based on two-part assertions of the form $\{P, C\}$, where P is a predicate on vector program variables, whereas C is a pure boolean parallel expression, also called *context expression*: in the current environment, this expression evaluates into the current activity context [5]. In this previous assertion language, the basic instance $\langle \forall u : X|_u = Y|_u + 1 \wedge \mathcal{C}|_u = (Y|_u = 2), \mathcal{C} \rangle$, is denoted by $\{\forall u : X|_u = Y|_u + 1, Y = 2\}$. There is no special variable denoting context. The main paradigm of this approach is that context is defined as a logical fonction from the environment. This assertion language is well-suited for writing proofs by annotations [3] (context expressions are the natural counterpart of where constructs), but it is insufficient to denote its own weakest liberal precondition [4]. Consider for instance the following weakest liberal precondition:

$$wlp(\mathsf{loop}\ True\ \mathsf{do}\ \mathsf{skip}, \{true, True\}),$$

which denotes all states (σ, c) from which the iteration diverges. In other words, it represents all states (σ, c) such that there is an index u with $c|_u = true$. The weakest liberal precondition is thus described by a property on the extent of parallelism. But context *cannot* be deduced from the environment part, because there is no link between context and environment in the final postcondition. As a consequence, there is no assertion to denote the weakest liberal precondition. The assertion language doesn't have the Definability Property.

Although some partial results have been proved ([3]), completeness of the initial proof system is not easy to establish.

Le Guyadec and Virot [10] proposed to restrict the assertion langage. They chose a more operational approach. They introduce a specific parallel variable $\sharp$ in the environment, which denotes the current context. A state in their new denotational semantics (denoted here by $[\![S]\!]_{GV}$ for a program S) is simply given by an environment σ. This semantics is such that, for all program S (variable $\sharp$ never appear in $Var(S)$) and environment σ,

$$[\![S]\!]_{GV}\ (\sigma) = \sigma' \qquad \text{iff} \qquad [\![S]\!](\sigma, \sigma(\sharp)) = (\sigma', \sigma'(\sharp)).$$

They thus define a one-part assertion language which has the definability property [10]. For instance, the weakest liberal precondition of our previous example (denoted here by wlp_{GV}) is well defined in their proof system:

$$wlp_{GV}\ (\mathsf{loop}\ True\ \mathsf{do}\ \mathsf{skip}, \{\forall u : \sharp|_u = true\}) = \{\exists u : \sharp|_u = true\}$$

The assertion language introduced in this paper is also a restriction of the initial one, but we didn't modify the initial denotational semantics of the $\mathcal{L}$ language. We follow a logical approach. By introducing a new variable ($\mathcal{C}$), we define an assertion language which has the definability property. This result illustrates a well known drawback of axiomatic semantics: this kind of semantics is not well suited to denote control flow properties of programs. It is particularly true in the case of parallel programs. For instance, Owicki and Gries introduce auxiliary variables to catch control flow information in the proof of parallel programs [11]. This enables them to prove properties like mutual exclusion.

It is easy to show that the approach of Le Guyadec and Virot is a special case of ours. Let S be a $\mathcal{L}$ program, Q a predicate and $W_{GV} = wlp_{GV}\ (S, \{Q\})$ in their semantics. Let $W = wlp(S, \langle Q \wedge \sharp = \mathcal{C}, \mathcal{C}\rangle)$ in our semantics. We have $\models W_{GV} \Leftrightarrow W[\sharp/\mathcal{C}]$.

Our proof system enables us to prove the completeness of the initial one. The complete proof can be found in [13]. We just sketch here the proof argumentation. Let S a $\mathcal{L}$ program, $\{Q, D\}$ an assertion in the original proof system and consider $\langle W', C'\rangle = wlp(S, \langle Q \wedge \forall u : D|_u = \mathcal{C}|_u, \mathcal{C}\rangle)$. From our definability result, there exists an assertion $\langle W', \mathcal{C}'\rangle$, where $\mathcal{C}'$ is such that $C' \notin Var(S)$. It is easy to show that $\models \{W', \mathcal{C}'\}S\{Q, D\}$. In [13], it is proved that $\vdash \{W', \mathcal{C}'\}S\{Q, D\}$. Another crucial observation is to see that, for all $\{P, C\}$ such that $\models \{P, C\}S\{Q, D\}$, we have a *logical consequence lemma*

$$\models P \wedge C = \mathcal{C} \Rightarrow W'.$$

We then can use the Consequence Rule and, as $\mathcal{C}$ is chosen such that $\mathcal{C} \notin (Var(P) \cup Var(Q) \cup Var(C) \cup Var(D))$, an *Auxiliary Variable Elimination Rule* finally yields $\vdash \{P, C\}S\{Q, D\}$.

7 Conclusion

We have presented a new type of assertions to prove programs written in the $\mathcal{L}$ language. We have built a proof system using this assertion language, and we have defined a weakest precondition calculus. We then proved the Definability Property and the completeness. We have shown that our approach shed a new light on the complexity of proving data-parallel programs. Proof systems for data-parallel languages are quite similar to those used in the case of scalar languages: they have the same structure and respect some crucial properties such as compositionality. However, they inherit additional complexity from the parallel world: introducing auxiliary variables is sometimes necessary to catch information about the control flow information, i.e. context evolution. It could be interesting to investigate this point in order to find out its logical significance in the proof process.

Acknowledgments

We thank anonymous referees for their useful comments.

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] J. De Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall, 1981.

[3] L. Bougé and D. Cachera. On the completeness of a proof system for a simple data-parallel language. In *Proc. 1st EuroPar Conf.*, LNCS, Stockholm, Sweden, August 1995.

[4] 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.

[5] 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.

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

[7] M. Clint and K.T. Narayana. On the completeness of a proof system for a synchronous parallel programming langage. In *Third Conference of FST/TCS*, Bangalore, India, December 1983.

[8] J. Gabarró and R. Gavaldà. An approach to correctness of data parallel algorithms. *Journal of Parallel and Distributing Computing*, (22), 1994.

[9] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communication of the ACM*, 12(10):576–580, 583, October 1969.

[10] Yann Le Guyadec and Bernard Virot. Sequential-like proofs of data-parallel programs. *Parallel Processing Letters*, 96. To appear.

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

[13] G. Utard. *Sémantique des langages à parallélisme de données. Applications à la validation et à la compilation.* PhD thesis, École Normale Supérieure de Lyon, December 1995.

[14] Glynn Winskel. *The Formal Semantics of Programming Languages : An introduction.* Foundations of Computing. The MIT Press, 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399