Proofs by annotations for a simple data-parallel language

Luc Bougé, David Cachera

Laboratoire de l'Informatique du Parallélisme

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

Proofs by annotations for a simple data-parallel language

Luc Bougé, David Cachera

March 1995

Research Report N° 95-08

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

Proofs by annotations for a simple data-parallel language

Luc Bougé, David Cachera

March 1995

Abstract

We present a proof outline generation system for a simple data-parallel kernel language called $\mathcal{L}$. We show that proof outlines are equivalent to the sound and complete Hoare logic defined for $\mathcal{L}$ in previous papers. Proof outlines for $\mathcal{L}$ are very similar to those for usual scalar-like languages. In particular, they can be mechanically generated backwards from the final post-assertion of the program. They appear thus as a valuable basis to implement a validation assistance tool for data-parallel programming.

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

Résumé

Nous présentons un système pour la génération de schémas de preuve par annotations *proof outlines* pour un petit noyau de langage à parallélisme de données appelé $\mathcal{L}$. Nous montrons que les schémas de preuve par annotations sont équivalents à la logique de Hoare pour le langage $\mathcal{L}$ définie dans les articles précédents. La manipulation des annotations des programmes $\mathcal{L}$ est très semblable à celle des langages scalaires habituels de type Pascal. En particulier, les annotations peuvent être générées automatiquement à partir de la post-condition du programme. Cette méthode constitue donc une base formelle intéressante pour l'implémentation d'outils d'aide à la programmation data-parallèle.

Mots-clés: Programmation parallèle, spécification et validation de programmes, sémantique des langages de programmation, langages data-parallèles, système de preuve, logique de Hoare, plus faibles préconditions.

Proofs by annotations for a simple data-parallel language

Luc Bougé*‡, David Cachera*

April 13, 1995

Abstract

We present a proof outline generation system for a simple data-parallel kernel language called $\mathcal{L}$. We show that proof outlines are equivalent to the sound and complete Hoare logic defined for $\mathcal{L}$ in previous papers. Proof outlines for $\mathcal{L}$ are very similar to those for usual scalar-like languages. In particular, they can be mechanically generated backwards from the final post-assertion of the program. They appear thus as a valuable basis to implement a validation assistance tool for data-parallel programming.

Contents

1 A sound and complete proof system for a small data-parallel language **2**
1.1 The $\mathcal{L}$ language 2
1.2 Denotational semantics of linear $\mathcal{L}$-programs 3
1.3 The $\vdash^*$ proof system 4
1.3.1 Assertion language 4
1.3.2 Proof system 5
1.4 Weakest preconditions calculus 7

2 A simple two-pass proof method **9**
2.1 First step: syntactic labeling 9
2.2 Second step: proof outline 10

3 A small example **11**

4 Equivalence of Proof Outlines and $\vdash^*$ **13**

5 Discussion **17**

*LIP, ENS Lyon, 46 Allée d'Italie, F–69364 Lyon Cédex 07, France.
‡Authors contact: Luc Bougé (`Luc.Bouge@lip.ens-lyon.fr`). This work has been partly supported by the French CNRS Coordinated Research Program on Parallelism, Networks and Systems PRS.

Introduction

Data-parallel languages have recently emerged as a major tool for large scale parallel programming. An impressive effort is currently being put on developing efficient compilers for High Performance Fortran (HPF). A data-parallel extension of C, primarily influenced by Thinking Machine's C*, is currently under standardization. Our goal is to provide all these new developments with the necessary semantic bases.

In previous papers, we have defined a simple, but representative, data-parallel kernel language [5], and we have described a natural semantics for it. We have designed a sound proof system based on an adaptation of Hoare logic [4]. We have shown it gives rise to a Weakest Precondition calculus [2], which can be used to prove its completeness for loop-free programs [3].

Yet, a crucial step remains to be done for a practical application of these results. Quoting Apt and Olderog's seminal book [1, Section 3.4]:

> *Formal proofs are tedious to follow. We are not accustomed to following a line of reasoning presented in small, formal steps [...].*
>
> *A possible strategy lies in the facts that [programs] are structured. The proof rules follow the syntax of the program, so the structure of the program can be used to structure the correctness proof. We can simply present the proof by giving a program with assertions interleaved at appropriate places [...].*
>
> *This type of proof is more simple to study and analyse than the one we used so far. Introduced by Gries and Owicki, it is called a Proof Outline.*

The presentation of Apt and Olderog focuses on control-parallel programs, that is, sequential processes composed with the $\|$ operator. In this paper, we show that the approach of Gries and Owicki can be adapted as well to data-parallel $\mathcal{L}$ programs, giving birth to a notion of data-parallel annotations.

For the sake of completeness, we briefly recall in Section 1 the definition of the $\mathcal{L}$ language, its logical two-part assertions, the associated Hoare logic and the Weakest Precondition calculus. Section 2 describes the formation rules for the Data-Parallel Proof Outlines. In contrast with the usual scalar case, they are generated in two passes. Pass 1 labels program instruction with their respective *extent of parallelism* (to be called *activity context* below); it works top-down. Pass 2 generates the intermediate assertions starting from the final post-condition; it works bottom-up. Section 3 describes an example. Section 4 proves our main result, which is the equivalence between this notion of Data-Parallel Proof Outline and the Hoare logic for $\mathcal{L}$.

1 A sound and complete proof system for a small data-parallel language

An extensive presentation of the $\mathcal{L}$ language can be found in [5]. For the sake of completeness, we briefly recall its denotational semantics as described in [2].

1.1 The $\mathcal{L}$ language

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* or the *extent of parallelism*. It can be seen as a boolean array where *true* denotes activity and *false* idleness.

The $\mathcal{L}$ language is designed as a common kernel of data-parallel languages like C* [9], HyperC [8] or MPL [7]. 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, i.e. expressions without side effects. 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 detail the syntax and semantics of such expressions any further. We introduce a special vector constant 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 $\mathcal{L}$-instructions are 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.

Conditioning: where B do S end. 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.

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 extent of parallelism is not modified.

In the following, we restrict ourselves to *linear* programs, i.e. programs without loops.

1.2 Denotational semantics of linear $\mathcal{L}$-programs

We recall the semantics of $\mathcal{L}$ defined in [2] 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 $true$. 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.

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

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 $[\![S]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c')$.

1.3 The $\vdash^*$ proof system

1.3.1 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. As we consider here only linear programs, S will always terminate. 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 [2]. 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 $\sigma(C) = c$. The set of states satisfying $\{P, C\}$ is denoted by $[\![\{P, C\}]\!]$. When no confusion may arise, we identify $\{P, C\}$ and $[\![\{P, C\}]\!]$.*

Definition 3 (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*

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

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 the definition of our assertion language. The usual Substitution Lemma [1] extends to this new setting.

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

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

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

Definition 4 (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)*

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

1.3.2 Proof system

We recall on Figure 1 the proof system defined in [3]. This system is a restricted proof system, in the sense that a number of rules 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.

Assignment Rule	$\dfrac{X \notin Var(D)}{\{Q[(D?E:X)/X], D\}\ X{:=}E\ \{Q,D\}}$
Communication Rule	$\dfrac{Y \notin Var(D)}{\{Q[(D?X\|_A : Y)/Y], D\}\ \textsf{get}\ X\ \textsf{from}\ A\ \textsf{into}\ Y\ \{Q,D\}}$
Sequencing Rule	$\dfrac{\{P,C\}\ S\ \{R,E\},\ \{R,E\}\ T\ \{Q,D\}}{\{P,C\}\ S;T\ \{Q,D\}}$
Conditioning Rule	$\dfrac{\{P, C \wedge B\}\ S\ \{Q,D\},\ Change(S) \cap Var(C) = \emptyset}{\{P,C\}\ \textsf{where}\ B\ \textsf{do}\ S\ \textsf{end}\ \{Q,C\}}$
Consequence Rule	$\dfrac{\{P,C\} \Rightarrow \{P',C'\},\ \{P',C'\}\ S\ \{Q',D'\},\ \{Q',D'\} \Rightarrow \{Q,D\}}{\{P,C\}\ S\ \{Q,D\}}$
Substitution Rule	$\dfrac{\{P,C\}\ S\ \{Q,D\},\quad Tmp \notin Var(S) \cup Var(Q) \cup Var(D)}{\{P[E/Tmp], C[E/Tmp]\}\ S\ \{Q,D\}}$

Figure 1: The $\vdash^*$ proof system for linear-$\mathcal{L}$

Definition 5 *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 $Change(S) \cap Var(D) = \emptyset$.

The proof system contains a particular rule, called the *Substitution Rule*. This rule is used to handle conditioning constructs where the variables appearing in the conditioning expression may be modified by the body of the construct. More formally, if we consider the program where B do S end with $Var(B) \cap Change(S) \neq \emptyset$, the value of B on exiting S may be different from its value on entering this body. This fact leads us to introduce *hidden variables*, i.e. variables that do not appear in programs, context expressions or postconditions. These variables are used to store temporarily the initial value of conditioning expressions and, as they do not appear in programs, these value remains unchanged during the execution of the body. As hidden variables are in a way "new" variables, there is no reason why they should appear in specifications. The role of the Substitution Rule is namely to get rid of them eventually.

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 1 (Soundness of $\vdash^*$ [3]) *The $\vdash^*$ proof system is sound: If $\vdash^* \{P,C\}\ S\ \{Q,D\}$, then $\models \{P,C\}\ S\ \{Q,D\}$.*

Construct	Conditions	Weakest Precondition
Assignment	$X \notin Var(D)$	$WP(X := E, \{Q, D\})$ $= \{Q[(D?E : X)/X], D\}$
Communication	$Y \notin Var(D)$	$WP(\mathsf{get}\ X\ \mathsf{from}\ A\ \mathsf{into}\ Y, \{Q, D\})$ $= \{Q[(D?X\|_A : Y)/Y], D\}$
Sequencing	—	$WP(S_1;S_2, \{Q, D\})$ $= WP(S_1, WP(S_2, \{Q, D\}))$
Conditioning (1)	$Var(D) \cap Change(S) = \emptyset$ $Var(B) \cap Change(S) = \emptyset$ $WP(S, \{Q, D \wedge B\}) = \{P, C\}$	$WP(\mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}, \{Q, D\})$ $= \{P, D\}$
Conditioning (2)	$Var(D) \cap Change(S) = \emptyset$ $Tmp \notin Var(S) \cup Var(Q) \cup Var(D)$ $WP(S, \{Q, D \wedge Tmp\}) = \{P, C\}$	$WP(\mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}, \{Q, D\})$ $= \{P[B/Tmp], D\}$

Figure 2: Definability properties of weakest preconditions for linear $\mathcal{L}$-programs

1.4 Weakest preconditions calculus

A weakest preconditions calculus has been presented in [2], and has been used to prove the completeness of the $\vdash^*$ proof system in [3]. We briefly recall here some useful definitions and results.

Definition 6 (Weakest preconditions) *Let $\mathcal{E}$ be a subset of State, S a linear $\mathcal{L}$-program. We define the* weakest preconditions *as*

$$WP(S, \mathcal{E}) = \{s \in State \mid [\![S]\!](s) \in \mathcal{E}\}$$

Lemma 2 (Consequence Lemma) $\models \{P, C\}\ S\ \{Q, D\}$ *iff* $[\![\{P, C\}]\!] \subseteq WP(S, \{Q, D\})$.

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. Definability results have been proved in [2]. They are listed up on Figure 2. We add here a general result on WP that will help us in the next section: if we use the Definability Properties to construct the assertion defining a weakest precondition, the variables appearing in this assertion already appear in the program, the postcondition or the context expression. In other words, and more intuitively, computing a WP doesn't generate "new" variables. This fact is expressed in the following proposition.

Proposition 1 *Let Z be a variable, S a program, Q an assertion and D a boolean expression such that $Var(D) \cap Change(S) = \emptyset$. If*

$$Z \notin Var(S) \cup Var(Q) \cup Var(D),$$

then there exists some assertion $\{P, C\}$ such that

$$WP(S, \{Q, D\}) = \{P, C\},$$

and

$$Z \notin Var(P) \cup Var(C).$$

Proof

This result is a consequence of the definability properties, and is established by induction on the structure of S.

- *If $S \equiv X{:=}E$, $WP(S, \{Q, D\}) = \{Q[(D?E : X)/X], D\}$. As $Z \notin \{X\} \cup Var(E) \cup Var(Q) \cup Var(D)$, Z doesn't appear in the weakest precondition.*
- *The case of communication is similar to that of assignment.*
- *If $S \equiv S_1;S_2$, then by induction hypothesis Z doesn't appear in the assertion $WP(S_2, \{Q, D\})$. As $WP(S_2, \{Q, D\})$ is used as postcondition for S_1, a second use of the induction hypothesis for S_1 shows that Z doesn't appear in the assertion $WP(S, \{Q, D\})$.*
- *If $S \equiv$ **where** B **do** T **end**, we have two cases to consider.*
  - *If $Var(B) \cap Change(S) = \emptyset$, we apply the first definability property for conditioning. Let us assume that $WP(T, \{Q, D \wedge B\}) = \{P, C\}$. We have $Z \notin Var(S)$, so $Z \notin Var(B)$. The induction hypothesis thus yields $Z \notin Var(P)$, so Z doesn't appear in $\{P, D\}$, which is the precondition for S.*
  - *If $Var(B) \cap Change(S) \neq \emptyset$, we apply the second definability property for conditioning. Let Tmp be a variable not in $Var(T) \cup Var(Q) \cup Var(D)$, and let $\{P, C\}$ be $WP(T, \{Q, D \wedge Tmp\})$. If $Z = Tmp$, then, as $WP(S, \{Q, D\}) = \{P[B/Tmp], D\}$, Z is substituted by B in the weakest precondition, so it doesn't appear in it any more. If $Z \neq Tmp$, then by induction hypothesis $Z \notin Var(P)$ and $Z \notin Var(B)$, so $Z \notin Var(P[B/Tmp])$.*

Proof of Proposition 1 is done. ❑

As shown in [3], the use of WP calculus is the key to establish the completeness of the $\vdash^*$ proof system.

Theorem 2 (Completeness of $\vdash^*$ [3]) *Let $\{P, C\}$ S $\{Q, D\}$ be a specification. If*

$$\models \{P, C\}\ S\ \{Q, D\}$$

then

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

2 A simple two-pass proof method

We present here a simple proof method that allows, after a first step that slightly transforms the program, to handle it as an usual scalar program. The first step consists in a labeling of the program that expresses the depth of conditioning constructs. In other words, a subprogram labeled by i is executed within the scope of i where constructs. This labeling follows the syntax of the program: labels are increased on entering the body of a new conditioning construct. Context expressions are saved here in a series of auxiliary variables. This allows us to alleviate any restriction on context expressions of conditioning constructs.

The second step consists in a proof method similar to that used in the scalar case. It is presented here in the form of a *proof outline*. As introduced by Gries and Owicki in 1976, this form gives a more convenient presentation of the proof, interleaving assertions and program constructs [1].

In this section, we give the formal description of the two steps, and then prove the equivalence between this proof method and the $\vdash^*$ proof system.

2.1 First step: syntactic labeling

In this step, we associate to each subprogram of the considered program an integer label that counts the number of nesting where constructs. Counting starts at 0 for the entire program. Consider for instance the program

```
where X>0 do
        X:=X+1;
        where X>2 do
                X:=X+1;
        end end
```

We want to get the following labeling.

```
(0) where X>0 do
        (1) X:=X+1;
        (1) where X>2 do
                (2) X:=X+1
        end
end
```

In order to store context expressions, we distinguish particular auxiliary variables that do not appear in programs.

Definition 7 *Variables* $\{Tmp_i \mid i \in \mathbb{N}\}$ *are such that for any program* S*, and for any index* i*,* $Tmp_i \notin Var(S)$*. This set is the set of* auxiliary variables.

The conditioning construct can be seen as a stack mechanism: entering a where construct is the same as pushing a value on a context stack, while exiting this construct corresponds to a “pop”. The label is namely the height of the stack. At a given point, the current context is corresponding to the conjunction of all the stack's values. Each auxiliary variable is used to store one cell of the context stack. Thanks to this storage, the variables appearing in context expressions may be modified. We thus can alleviate restrictions on context expressions of conditioning constructs.

For a subprogram at depth i, the current context is the current value of $Tmp_0 \wedge \ldots \wedge Tmp_i$. To get a clearer presentation of this fact, we add annotations of the form $[Tmp_i \equiv B]$ to each where construct. The previous example is recast into

```
(0) where X>0 do [Tmp_1 ≡ X > 0]
        (1) X:=X+1;
        (1) where X>2 do [Tmp_2 ≡ X > 2]
                (2) X:=X+1
        end
end
```

We now give a formal definition of program labeling. It is made by induction on the program's syntactic structure, and expressed by the rules listed below, $\varphi(\mathsf{S}, 0)$ being the labeling of program S.

$$\begin{array}{rcl}
\varphi(\ X := E\ , i) & = & (i)\ X := E \\
\varphi(\ \mathsf{get}\ X\ \mathsf{from}\ A\ \mathsf{into}\ Y\ , i) & = & (i)\ \mathsf{get}\ X\ \mathsf{from}\ A\ \mathsf{into}\ Y \\
\varphi(\ S\ ;\ T\ , i) & = & \varphi(S, i)\ ;\ \varphi(T, i) \\
\varphi(\ \mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{end}\ , i) & = & (i)\ \mathsf{where}\ B\ \mathsf{do}\ [Tmp_{i+1} \equiv B] \\
 & & \qquad \varphi(S, i+1) \\
 & & \qquad \mathsf{end}
\end{array}$$

2.2 Second step: proof outline

A proof outline is a visual and convenient way to present a proof with assertions interleaved in the text of the program at appropriate places [1]. The structure of the proof follows the structure of the program, thus giving a more readable presentation.

As we use labeled programs, and auxiliary variables to store contexts, we know at each place in the program the expression denoting the current context. We then can drop context expressions out of assertions and proceed exactly the same way as in the scalar case, with backward substitutions. The only differences are that expressions in substitutions are conditioned by a conjunction of Tmp_k and that the data-parallel where construct adds a new substitution. The rules for inserting assertions in proof outlines are given below. Contiguity between two assertions refers to the use of the consequence rule. If S is a labeled subprogram, we denote by S^* a proof outline obtained from S by insertion of assertions, and by $Lab(S)$ the label associated to S.

Notice that, as labeling starts at 0 for the entire program, Tmp_0 thus denotes the initial context in which S is executed.

$$\frac{\forall j > i,\ Tmp_j \notin Var(Q)}{\{Q[\bigwedge_{k=0}^{i} Tmp_k ? E : X/X]\}\ (i)\ \mathsf{X := E}\ \{Q\}}$$

$$\frac{\forall j > i,\ Tmp_j \notin Var(Q)}{\{Q[\bigwedge_{k=0}^{i} Tmp_k ? X|_A : Y/Y]\}\ (i)\ \mathsf{get\ X\ from\ A\ into\ Y}\ \{Q\}}$$

$$\frac{\{P\}\ S^*\ \{R\} \quad \{R\}\ T^*\ \{Q\} \quad \forall j > Lab(S),\ Tmp_j \notin Var(R) \cup Var(Q)}{\{P\}\ S^*;\{R\}\ T^*\ \{Q\}}$$

$$\frac{P \Rightarrow P' \quad \{P'\}\ S^*\ \{Q'\} \quad Q' \Rightarrow Q \quad \forall j > Lab(S), Tmp_j \notin Var(Q) \cup Var(Q')}{\{P\}\{P'\}\ S^*\ \{Q'\}\{Q\}}$$

$$\frac{\{P\}\ S^*\ \{Q\} \quad Lab(S) = i+1 \quad \forall j > i, Tmp_j \notin Var(Q)}{\{P[B/Tmp_{i+1}]\}\ (i)\ \textsf{where B do}\ [Tmp_{i+1} \equiv B]\ \{P\}\ S^*\ \{Q\}\ \textsf{end}\{Q\}}$$

$$\frac{\{P\}\ S^*\ \{Q\}}{\{P\}\ S^{**}\ \{Q\}}$$

where S^{**} is obtained from S^* by deleting any assertion.

Let us explain intuitively the need of restrictions of the form "$\forall j > i, Tmp_j \notin Var(Q)$". In the rule for the conditioning construct, we substitute Tmp_{i+1} by B. We thus need that $Tmp_{i+1} \notin Var(Q)$ to respect the conditions of the Substitution Rule. But, as the postcondition (Q) is the same for S and for **where** B **do** S **end**, we need that condition to be satisfied for every nesting depth greater than $Lab(S)$.

3 A small example

We go back in this section to our previous example. We want to prove the two following specifications.

```
{X|u = 2, True}
where X>0 do
        X:=X+1 ;
        where X>2 do
                X:=X+1
                end
        end
{X|u = 4, True}
```

```
{X|u = 1, True}
where X>0 do
        X:=X+1 ;
        where X>2 do
                X:=X+1
                end
        end
{X|u = 2, True}
```

The proofs are simply done by establishing the following proof outline — the result of the first step has already been given as example in the previous section.

First proof $\{(Tmp_0 \wedge X > 0 \wedge (Tmp_0 \wedge X > 0 ? X+1 : X) > 2 ? (Tmp_0 \wedge X > 0 ? X+1 : X) + 1 : (Tmp_0 \wedge X > 0 ? X+1 : X))|_u = 4\}$

(0) where X>0 do $[Tmp_1 \equiv X > 0]$

$\{(Tmp_0 \wedge Tmp_1 \wedge (Tmp_0 \wedge Tmp_1?X+1:X) > 2?(Tmp_0 \wedge Tmp_1?X+1:X)+1:$
$(Tmp_0 \wedge Tmp_1?X+1:X))|_u = 4\}$

(1) X:=X+1 ;

$\{(Tmp_0 \wedge Tmp_1 \wedge X > 2?X+1:X)|_u = 4\}$

(1) where X>2 do $[Tmp_2 \equiv X > 2]$

$\{(Tmp_0 \wedge Tmp_1 \wedge Tmp_2?X+1:X)|_u = 4\}$

(2) X:=X+1

$\{X|_u = 4\}$

end

$\{X|_u = 4\}$

end

$\{X|_u = 4\}$

If we denote by P the first assertion of this proof outline, we only have to prove that

$$X|_u = 2 \wedge Tmp_0 = True \Rightarrow P.$$

In other words, we prove that

$$X|_u = 2 \Rightarrow P[True/Tmp_0]$$

The assertion $P[True/Tmp_0]$ is equivalent to

$$\{(X > 0 \wedge (X > 0?X+1{:}X) > 2?(X > 0?X+1{:}X)+1{:}(X > 0?X+1{:}X))|_u = 4\}$$

Let us consider an index u such that $X|_u = 2$. Then, the boolean expression $(X > 0)|_u$ is true. As $X+1|_u > 2$, $((X > 0?X+1:X) > 2)|_u$ is also true.

Conditional expression

$$(X > 0 \wedge (X > 0?X+1:X) > 2?(X > 0?X+1:X)+1:(X > 0?X+1:X))|_u$$

thus simplifies into $(X > 0?X+1:X)+1|_u$, which in turn simplifies into $X+1+1|_u$.

Assertion $P[True/Tmp_0]$ thus simplifies into $X+1+1|_u = 4$, which is true.

Second proof. As no simplification using the value of X occurs in the first proof outline, the second is almost the same: we just replace the value 4 by the value 2. Then, if we denote by P' the assertion obtained by substituting 4 by 2 in P, we just have to check that

$$X|_u = 1 \Rightarrow P'[True/Tmp_0]$$

Let us consider an index u such that $X|_u = 1$. Then, the boolean expression $(X > 0)|_u$ is true. But this time, as $X + 1|_u = 2$, $((X > 0?X + 1 : X) > 2)|_u$ is false.

Conditional expression

$$(X > 0 \wedge (X > 0?X + 1 : X) > 2?(X > 0?X + 1 : X) + 1 : (X > 0?X + 1 : X))|_u$$

thus simplifies into $(X > 0?X + 1 : X)|_u$, which in turn simplifies into $X + 1|_u$.

Assertion $P'[\mathit{True}/\mathit{Tmp}_0]$ thus simplifies into $X + 1|_u = 2$, which is true.

4 Equivalence of Proof Outlines and $\vdash^*$

We now want to prove that the method defined above is equivalent to the $\vdash^*$ proof system. More precisely, we want to prove the following theorem.

Theorem 3 *Let $\{P\}$ $(0)S$ $\{Q\}$ be a formula such that for each $j > 0$, $\mathit{Tmp}_j \notin \mathit{Var}(Q)$.*

$$\{P\}\ S^*\ \{Q\} \textit{is a proof outline for } S$$

$$\Updownarrow$$

$$\vdash^* \{P, \mathit{Tmp}_0\}\ S\ \{Q, \mathit{Tmp}_0\}$$

We actually prove the more general following fact.

Proposition 2 *Let S be a subprogram labeled by i, and P and Q assertions such that $\forall j > i$, $\mathit{Tmp}_j \notin \mathit{Var}(Q)$. Then*

$$\{P\}\ S^*\ \{Q\}$$

is a proof outline for S if and only if

$$\vdash^* \{P, \mathit{Tmp}_0 \wedge \ldots \wedge \mathit{Tmp}_i\}\ S\ \{Q, \mathit{Tmp}_0 \wedge \ldots \wedge \mathit{Tmp}_i\}$$

We begin with the easiest part of the proof: if there exists a proof outline, then the desired specification is derivable in $\vdash^*$.

Proof ____

Let S be a subprogram labeled with i, and $\{P\}$ S^* $\{Q\}$ a proof outline for S. The proof is by induction on the length of the construction made to obtain the proof outline. We have six cases to consider, corresponding respectively to each derivation rule for proof outlines.

- *If the last rule applied was*

$$\frac{\forall j > i, \mathit{Tmp}_j \notin \mathit{Var}(Q)}{\{Q[\bigwedge_{k=0}^{i} \mathit{Tmp}_k?E : X/X]\}\ (i)\ \mathsf{X} := \mathsf{E}\ \{Q\}},$$

  then, since $X \notin \{\mathit{Tmp}_i \mid i \in \mathbb{N}\}$, we have $\vdash^* \{P, \mathit{Tmp}_0 \wedge \ldots \wedge \mathit{Tmp}_i\}\ S\ \{Q, \mathit{Tmp}_0 \wedge \ldots \wedge \mathit{Tmp}_i\}$.

- *The second case, dealing with the communication statement, is handled exactly the same way.*

- *If the last rule applied was*

$$\frac{P \Rightarrow P' \quad \{P'\}\ S^*\ \{Q'\} \quad Q' \Rightarrow Q \quad \forall j > Lab(S), Tmp_j \notin Var(Q) \cup Var(Q')}{\{P\}\{P'\}\ S^*\ \{Q'\}\{Q\}},$$

  then by induction hypothesis we have $\vdash^* \{P', Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q', Tmp_0 \wedge \ldots \wedge Tmp_i\}$*, so the consequence rule of* $\vdash^*$ *applies and gives the desired result.*

- *If the last rule applied was the rule for sequential composition, then there exist* S_1 *and* S_2 *such that* $S = S_1;S_2$*, and an assertion* R *such that we have the proof outlines* $\{P\}\ S_1^*\ \{R\}$ *and* $\{R\}\ S_2^*\ \{Q\}$*. Furthermore, we know that* S_1 *and* S_2 *are labeled by the same value* i*. By the rule for sequential composition in proof outlines, we have* $\forall j > i, Tmp_j \notin Var\ R$*. By induction hypothesis, we thus have*

$$\vdash^* \{P, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S_1^*\ \{R, Tmp_0 \wedge \ldots \wedge Tmp_i\}$$

  and

$$\vdash^* \{R, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S_2^*\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

  Then, the Sequencing Rule of $\vdash^*$ *applies and yields*

$$\vdash^* \{P, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

- *If the last used rule was*

$$\frac{\{P'\}\ T^*\ \{Q\} \quad Lab(T) = i+1 \quad \forall j > i, Tmp_j \notin Var(Q)}{\begin{array}{l}\{P'[B/Tmp_{i+1}]\}\ (i)\ \textsf{where}\ B\ \textsf{do}\ [Tmp_{i+1} \equiv B] \\ \qquad\qquad\qquad\qquad\qquad T^* \\ \qquad\qquad\qquad\qquad\qquad \textsf{end}\{Q\}\end{array}}$$

  with $P = P'[B/Tmp_{i+1}]$*. We have* $\forall j > i+1, Tmp_j \notin Var(Q)$*, so by induction hypothesis*

$$\vdash^* \{P', Tmp_0 \wedge \ldots \wedge Tmp_i \wedge Tmp_{i+1}\}\ T\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i \wedge Tmp_{i+1}\}$$

  As $\{P' \wedge Tmp_{i+1} = B, Tmp_0 \wedge \ldots \wedge Tmp_i \wedge B\} \Rightarrow \{P', Tmp_0 \wedge \ldots \wedge Tmp_i \wedge Tmp_{i+1}\}$*, the Consequence Rule yields*

$$\vdash^* \{P' \wedge Tmp_{i+1} = B, Tmp_0 \wedge \ldots \wedge Tmp_i \wedge B\}\ T\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i \wedge Tmp_{i+1}\}.$$

  The **where** *Rule applies and yields*

$$\vdash^* \{P' \wedge Tmp_{i+1} = B, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

  Finally, using the Substitution Rule with B/Tmp_{i+1} *yields*

$$\vdash^* \{P, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

- *The last case (elimination of assertions in the proof outline) is straightforward.*

The proof of the first part of Proposition 2 is done. ❑

We now want to prove second part of Proposition 2. The proof uses the weakest preconditions and needs the following auxiliary result.

Proposition 3 *Let Q be an assertion such that $Tmp_{i+1} \notin Var(Q)$. If*

$$WP(S, \{Q, Tmp_0 \wedge \ldots \wedge Tmp_{i+1}\}) = \{P, Tmp_0 \wedge \ldots \wedge Tmp_{i+1}\},$$

then

$$WP(\textsf{where } B \textsf{ do } S \textsf{ end}, \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}) = \{P[B/Tmp_{i+1}], Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

Proof

Let $(\sigma, c) \in WP(\textsf{where } B \textsf{ do } S \textsf{ end}, \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\})$. Let (σ', c) be $[\![\textsf{where } B \textsf{ do } S \textsf{ end}]\!](\sigma, c)$. We have $[\![S]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c \wedge \sigma(B))$, and $(\sigma', c) \models \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}$ by the definition of WP. Let $\sigma_1 = \sigma[Tmp_{i+1} \leftarrow \sigma(B)]$, and $\sigma_1' = \sigma_1[Tmp_{i+1} \leftarrow \sigma(B)]$. Since Tmp_{i+1} is an auxiliary variable, we have $Tmp_{i+1} \notin Var(S)$, and

$$[\![S]\!](\sigma_1, c \wedge \sigma(B)) = (\sigma_1', c \wedge \sigma(B)),$$

and, as $Tmp_{i+1} \notin Var(Q)$,

$$(\sigma_1', c) \models \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

Furthermore, $\sigma_1'(Tmp_{i+1}) = \sigma(B)$, so

$$(\sigma_1', c \wedge \sigma(B)) \models \{Q, Tmp_0 \wedge \ldots \wedge Tmp_{i+1}\}.$$

We can deduce that $(\sigma_1, c \wedge \sigma(B)) \models \{P, Tmp_0 \wedge \ldots \wedge Tmp_{i+1}\}$. Thus

$$\sigma \models P[B/Tmp_{i+1}].$$

As Tmp_i is an auxiliary variable, we have $\forall i, Tmp_i \notin Var(S)$, so $\sigma'(Tmp_0 \wedge \ldots \wedge Tmp_i) = c$ implies

$$\sigma(Tmp_0 \wedge \ldots \wedge Tmp_i) = c.$$

Conversely, let $(\sigma, c) \in [[\{P[B/Tmp_{i+1}], Tmp_0 \wedge \ldots \wedge Tmp_i\}]]$, and $\sigma_1 = \sigma[Tmp_{i+1} \leftarrow \sigma(B)]$. We have

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

with $[\![S]\!](\sigma, c \wedge \sigma(B)) = (\sigma', c \wedge \sigma(B))$.

If $\sigma_1' = \sigma_1[Tmp_{i+1} \leftarrow \sigma(B)]$, we also have

$$[\![\textsf{where } B \textsf{ do } S \textsf{ end}]\!](\sigma_1, c) = (\sigma_1', c),$$

with $[\![S]\!](\sigma_1, c \wedge \sigma(B)) = (\sigma_1', c \wedge \sigma(B))$.

As $(\sigma, c) \in [[\{P[B/Tmp_{i+1}], Tmp_0 \wedge \ldots \wedge Tmp_i\}]]$, $\sigma_1 \models P$, and as $Tmp_{i+1} \notin Var(B)$, we have $\sigma_1(Tmp_0 \wedge \ldots \wedge Tmp_{i+1}) = c \wedge \sigma(B)$. By hypothesis, we have thus

$$(\sigma_1', c \wedge \sigma(B)) \models \{Q, Tmp_0 \wedge \ldots \wedge Tmp_{i+1}\}.$$

As $Tmp_{i+1} \notin Var(Q)$, we conclude that

$$\sigma' \models Q$$

Furthermore, $\forall i, Tmp_i \notin Var(S)$, so

$$\sigma'(Tmp_0 \wedge \ldots \wedge Tmp_i) = \sigma(Tmp_0 \wedge \ldots \wedge Tmp_i) = c.$$

This concludes the proof of proposition 3. ❑

We can now prove the second part of Proposition 2.

Proof

Let us assume that

$$\vdash^* \{P, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

We want to find a proof outline of the form

$$\{P\}\ S^*\ \{Q\}.$$

We construct this outline by induction on the structure of S.

- *If $S \equiv X{:=}E$: by the soundness of the proof system, we have*

$$\models \{P, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

  By the definition of WP, we have

$$\{P, Tmp_0 \wedge \ldots \wedge Tmp_i\} \Rightarrow WP(S, \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\})$$

  where $WP(S, \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}) = \{Q[Tmp_0 \wedge \ldots \wedge Tmp_i?E : X/X], Tmp_0 \wedge \ldots \wedge Tmp_i\}$. Then

$$\begin{array}{ll} \{P\} & \\ \{Q[Tmp_0 \wedge \ldots \wedge Tmp_i?E : X/X]\} & \\ & S \\ \{Q\} & \end{array}$$

  is a proof outline for S.

- *The case of communication statement is handled the same way.*
- *If $S \equiv S_1;S_2$. Let*

$$\{P_2, Tmp_0 \wedge \ldots \wedge Tmp_i\} = WP(S_2, \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\})$$

  and

$$\{P_1, Tmp_0 \wedge \ldots \wedge Tmp_i\} = WP(S_1, \{P_2, Tmp_0 \wedge \ldots \wedge Tmp_i\}).$$

  As $\forall j > i$, $Tmp_j \notin Var(S) \cup Var(Q)$, Lemma 1 guarantees that $\forall j > i$, $Tmp_j \notin Var(P_2)$. The premises of the rule for sequential composition are thus satisfied. By the soundness of $\vdash^*$, we have $\models \{P, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}$, so by the definition of WP,

$$P \Rightarrow P_1.$$

  Then

$$\begin{array}{ll} \{P\} & \\ \{P_1\} & \\ & S_1 \\ \{P_2\} & \\ & S_2 \\ \{Q\} & \end{array}$$

  is a proof outline for S.

- *Consider now the case when* $S \equiv$ ***where*** B ***do*** T ***end***. *The weakest preconditions calculus enables us to construct a proof*

$$\vdash^* \{P', Tmp_0 \wedge \ldots \wedge Tmp_{i+1}\}\ T\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_{i+1}\},$$

where

$$\{P', Tmp_0 \wedge \ldots \wedge Tmp_{i+1}\} = WP(T, \{Q, Tmp_0 \wedge \ldots \wedge Tmp_{i+1}\}).$$

By induction hypothesis,

$$\begin{array}{l} \{P'\} \\ \qquad T^* \\ \{Q\} \end{array}$$

is a proof outline for T.
But Proposition 3 yields

$$WP(S, \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}) = \{P'[B/Tmp_{i+1}], Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

Then, by the soundness of the proof system, we have

$$\models \{P, Tmp_0 \wedge \ldots \wedge Tmp_i\}\ S\ \{Q, Tmp_0 \wedge \ldots \wedge Tmp_i\}.$$

We conclude that $P \Rightarrow P'[B/Tmp_{i+1}]$ *and that*

$$\begin{array}{l} \{P\} \\ \{P'[B/Tmp_{i+1}]\} \\ \qquad \textbf{\textit{where}}\ B\ \textbf{\textit{do}}\ [Tmp_{i+1} \equiv B] \\ \qquad\qquad \{P'\} \\ \qquad\qquad T^* \\ \qquad\qquad \{Q\} \\ \qquad\qquad \textbf{\textit{end}} \\ \{Q\} \end{array}$$

is a proof outline for S.

❑

5 Discussion

We have defined a notion of Proof Outline for a simple data-parallel kernel language. Due to the two-part nature of the program assertions, it works in two passes. Pass 1 labels labels each instruction with its respective extent of parallelism top-down; Pass 2 generates the intermediate annotations bottom-up, starting from the final post-condition.

Pass 1 amounts to a simple rewriting. It could easily be handled by some advanced text editor. The rewriting process is slightly more complex due to the possible conflict between the vector boolean expressions denoting the current extent of parallelism and the assignments. Fresh temporary variables Tmp_i have to be introduced to save the activity contexts. Pass 2 is very similar to a Proof Annotation generating system for usual, scalar Pascal-like languages. The only difference lies in the slightly more complex substitution mechanism.

This similarity confirms that validating data-parallel programs is of the same level of complexity as validating scalar programs. This is in strong contrast with control-parallel CSP-like programs.

In this respect, the data-parallel programming model appears as a suitable basis for large-scale parallel software engineering.

A number of additional remarks can be made.

- Our equivalence result could probably be adapted to other shapes of assertions. It could be interesting to consider for instance the one-part assertions of Le Guyadec and Virot [6] where the current extent of parallelism is kept as the value of a special $\sharp$ symbol.

- Our two-pass annotation method could easily be carried out mechanically and integrated in some design/validation assistance tool. The main difficulty lies in keeping the assertions simple enough to be understood (and corrected!) by a human reader. The complex substitution mechanism generates nested conditional expressions which should be simplified on the fly by some additional tool.

- Consider a conditioned statement (i) where B do S. If the conditioned body S does not interfere with the expression denoting the current extent of parallelism, there is no need to introduce any auxiliary Tmp_{i+1} variable. One can as well use the conditioning expression B directly. This will probably result in simpler assertions. Such an optimization should definitely be considered in designing any *real* assistance tool.

- Proof outlines can also be used for automatic program documentation. An interesting application would be to generate annotations at certain "hot spots" in the program only, focusing on a set of crucial program variables. This could probably serve as a basis for an interactive tool where the user could build at the same time *both* the program and a (partial) proof of it.

References

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

[2] L. Bougé, Y. Le Guyadec, G. Utard, and B. Virot. On the expressivity of a weakest preconditions calculus for a simple data-parallel programming language. In *ConPar'94–VAPP VI*, Linz, Austria, September 1994.

[3] 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.

[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 Computing Systems*, 8:363–378, 1992.

[6] Y. Le Guyadec, B. Virot. Axiomatic semantics of conditioning constructs and non-local control transfers in data-parallel languages. Research Report 94–15, LIFO, Orléans, France, 1994.

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

[8] N. Paris. HyperC specification document. Technical Report 93–1, HyperParallel Technologies, 1993.

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