HAL
open science

A Logical Framework to Prove Properties of Alpha Programs

Luc Bougé, David Cachera

▶ To cite this version:

Luc Bougé, David Cachera. A Logical Framework to Prove Properties of Alpha Programs. [Research Report] RR-3031, INRIA. 1996. inria-00073662

HAL Id: inria-00073662
https://inria.hal.science/inria-00073662v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A logical framework to prove properties of ALPHA *programs*

Luc Bougé, David Cachera

N° 3031

Novembre 1996

THÈME 1

Rapport de recherche

ISSN 0249-6399

A logical framework to prove properties of ALPHA programs

Luc Bougé*, David Cachera*

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n° 3031 — Novembre 1996 — 20 pages

Abstract: We present an assertional approach to prove properties of ALPHA programs. ALPHA is a functional language based on affine recurrence equations. We first present two kinds of operational semantics for ALPHA together with some equivalence and confluence properties of these semantics. We then present an attempt to provide ALPHA with an external logical framework. We therefore define a proof method based on invariants. We focus on a particular class of invariants, namely canonical invariants, that are a logical expression of the program's semantics. We finally show that this framework is well-suited to prove equivalence properties between ALPHA programs.

Key-words: Concurrent programming, recurrence equations, specifying and verifying and reasoning about programs, semantics of programming languages, data-parallel languages, proof methodology, invariants.

(Résumé : tsvp)

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

* Contact: {Luc.Bouge,David.Cachera}@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

Un cadre logique pour la preuve de programmes ALPHA

Résumé : Nous présentons une méthode de preuve par assertions pour les programmes ALPHA. ALPHA est un langage fonctionnel d'équations récurrentes affines. Nous présentons tout d'abord deux types de sémantiques opérationnelles pour ALPHA, ainsi que des propriétés d'équivalence et de confluence de ces sémantiques. Nous munissons ensuite ALPHA d'un cadre logique externe au langage. Nous définissons pour cela une méthode de preuve fondée sur l'utilisation d'invariants. Nous insistons sur une classe particulière d'invariants, les invariants canoniques. Nous montrons finalement que ce cadre est particulièrement adapté à la preuve d'équivalence de programmes ALPHA.

Mots-clé : Programmation parallèle, équations récurrentes, spécification et validation de programmes, sémantique des langages de programmation, langages data-parallèles, méthode de preuve, invariants.

Introduction

The ALPHA programming language has been designed by Quinton and his group [3] as a basis to design, combine, optimize and eventually compile systolic algorithms. An ALPHA program is a set of affine recurrence equations (ARE) on multidimensional polyhedric integer domains. ALPHA is a strict, strongly typed functional language, restricted to a very simple form of mutual recursion, and it has proved to be an unvaluable tool in this field.

Usual manipulations on systolic algorithms can be seen as textual syntactic rewritings on the text of ALPHA programs. Parallelization amounts to reindexing the equations through a change of basis. Optimization amounts to add, delete and combine equations and variables according to a number of equivalence-preserving rules. Finally, compiling an ALPHA program amounts to eventually rewrite it into such a simple form that each equation corresponds to some elementary logical operator, each indexing to some hardware connection, and one of the component of the underlying basis to the hardware clock.

Much work has therefore been devoted to studying such transformations, and designing tools to assist the user in applying them. Yet, the problem of abstractly *proving* properties of ALPHA programs has been somewhat overlooked. In fact, a way of checking a property of an ALPHA program is to transform it into so simple a form that the satisfaction of intended property is somehow "obvious". But, the transformation process can be extremely complex without any common magnitude with the property under study. Also, transformations guarantee the exact equivalence between the original program and its transformed form, whereas much of it may be irrelevant for the property under study. Our claim is therefore that there is room here for an alternative approach to proofs, based on an *external logical framework* instead of an *internal transformational one*.

The situation is reminiscent of the one prevailing in the 70's and 80's in the field of program verification. Two alternative directions were competing to prove sequential programs correct. 1) Internal: refine the program up to equivalence, so that the intended property is eventually made explicit. 2) External: equip the program with some kind of logical assertions, so that the proof of correctness boils down to checking their abstract mutual consistency. For Pascal-like imperative programs, the balance is clearly in favor of the external approach (Floyd's assertions, Hoare's logic, Gries-Owicki method, etc.). No other alternative may develop, as the transformational calculus of such imperative programs is so poor. In contrast, more abstract frameworks such as Unity or CCS present a balance picture: the two approaches co-exist and fruitfully interact. The ALPHAlanguage benefits from a rich and moreover semi-automatizable transformational calculus, so that the internal approach has been almost exclusively developed.

Yet, the internal approach is used informally in several papers such as [5] where an ALPHA program is proved to be equivalent to its closed form, and [1] where a logical induction argument is used to prove a refinement step. The goal of this paper is precisely to set up an external framework in which these "handwaving" manipulations can be expressed and justified. Moreover, we show that this framework can give account of a number of methods to prove the equivalence of ALPHA programs, and open new direction in the search for useful internal textual transformations.

The paper is organized as follows. We briefly recall the essentials of the ALPHA language for self-containedness, and we discuss its operational semantics as a basis to interprete logical formulae. Then, we define our logical framework and show its completeness: all correct assertions can be proved. Finally, we show that this framework can give account of some pragmatic methods used so far. The discussion sums up our contribution and lists perspectives.

1 The ALPHA language

The ALPHA language has been initially designed by Mauras [3] for the synthesis of regular architectures. It is based on the model of *recurrence equations*, introduced by Karp, Miller and Winograd [2]. In this section, we briefly introduce the principles and notations of the ALPHA language. See [6] for more details. In the following, we denote by $\mathbb{Z}$ and $\mathbb{Q}$ the sets of integers and rational numbers, respectively.

1.1 Domains and variables

All the data structures manipulated in ALPHA are defined over polyhedral domains. We give the following definitions.

Definition 1 (Polyhedron) *A* polyhedron $\mathcal{P}$ *of dimension n is a subspace of $\mathbb{Q}^n$ bounded by a finite number of hyperplanes.*

Definition 2 (Domain) *A polyhedral domain $\mathcal{D}$ of dimension n is defined as*

$$\mathcal{D} = \mathbb{Z}^n \cap \mathcal{P},$$

where $\mathcal{P}$ is a polyhedron.

Definition 3 (Data field) *A data field is a mapping from a domain to values in a given data type.*

$$X \ : \ \mathcal{D} \longrightarrow \mathcal{T}$$

where $\mathcal{T}$ is either `integer`, `boolean`, *or* `real`.

remark: *Data fields have been historically called* variables *in* ALPHA. *In this paper, we denote by variable a single point (instance) of a data field.*

remark: *Scalar data fields are associated to the trivial domain $\mathbb{Z}^0$.*

1.2 Programs

ALPHA is a strongly typed, *structured equational* language. A system (program) in ALPHA consists of

- the declaration of input data fields;
- the declaration of output data fields;
- the declaration of local data fields;
- a set of affine recurrence equations, delimited by the keywords `let` and `tel`, defining output and local data fields.

As an example, we give a program that implements a convolution filter, taken from [6]. The output of this program is computed as $y[i] = \sum_{j=1}^{4} a[j] * x[i-j+1]$.

```
system convolution( a : { j | 1<=j<=4 } of integer;
                    x : { i | i>=1 } of integer )
       returns ( y : { i | i>=4 } of integer; )
    var
       Y : { i,j | 0<=j<=4 ; i>=4 } of integer;
    let
       Y[i,j] = case
                { | j=0 } : 0;
                { | 1<=j<=4 } : Y[i,j-1] + a[j] * x[i-j+1];
                esac;
       y[i] = Y[i,4];
    tel
```

In this example, expressions such as `{ j | 1<=j<=4 }` denote domains, `X` and `a` are the input data fields, `y` is the output data field and `Y` is a local data field. Notice that a program may have several output data fields.

Due to the equational nature of the language, ALPHA programs respect the *substitution principle*: any instance of a data field in any expression may be replaced by the right hand side of the equation defining that data field. Moreover, ALPHA is a single assignment language, and the equations in a system are unordered.

1.3 Operators

The expressions appearing on the right hand side of equations are obtained by combining data fields or constants by means of two kinds of operators: *pointwise* operators and *spatial* operators.

1.3.1 Pointwise operators

These operators are the classical data-parallel componentwise generalizations of scalar operators. For instance, $X + Y$ denotes the mapping

$$\begin{array}{rccc} X+Y & : & \mathcal{D}_x \cap \mathcal{D}_y & \longrightarrow & \mathcal{T} \\ & & i & \longmapsto & X_i + Y_i \end{array}$$

where $\mathcal{D}_x$ and $\mathcal{D}_y$ are the domains of X and Y, X_i and Y_i their values at index i, and $\mathcal{T}$ their common data type.

The domain of an expression defined by means of a pointwise operator is implicitely the intersection of the domains of the subexpressions combined by that operator.

1.3.2 Spatial operators

These operators are used to manipulate domains. We have three spatial operators, namely the *dependence, restriction* and *case* operators.

- The dependence operator. A *dependence function* is a mapping from $\mathbb{Z}^n$ to $\mathbb{Z}^m$, where $n, m \in \mathbb{N}$. It is denoted by $(i, j, \ldots \longrightarrow f(i, j, \ldots))$, where f is an affine function. For instance, $(i, j \longrightarrow i - j + 1)$ is a mapping from $\mathbb{Z}^2$ to $\mathbb{Z}$.

  The *dependence operator* "." combines an expression and a dependence function. If E is an expression and d a dependence function, $E.d$ is the composition of the mapping denoted by expression E with the affine function denoted by d. For instance, $X.(i, j \longrightarrow j, i)$ is the transpose of a two-dimensional data field X. The domain of $E.d$ is the preimage of the domain $\mathcal{D}$ of E by d.

  Two special cases arise when n or m is 0. For instance, $0.(i, j \longrightarrow)$ extends the constant 0 to $\mathbb{Z}^2$. Conversely, $(\longrightarrow k)$, where k is a constant, denotes a mapping from $\mathbb{Z}^0$ to $\mathbb{Z}$.

- The restriction operator. It restricts the domain of an expression by means of affine constraints. For instance, $\{i, j \mid i \geq 0\} : X$ restricts mapping X to the intersection of the domain of X with the half-plane $i \geq 0$.

- The case operator. It pieces together a set of disjoint subexpressions. The value of expression `case` $E_1; \ldots E_n;$ `esac` at index i is the value of expression E_k if i is in the domain of E_k. To be valid, the branches of the `case` must have disjoint domains. The resulting domain is the union of the domains of all subexpressions.

2 Two kinds of operational semantics

ALPHA in its definition does not specify the order of evaluation for expressions. A program may have several evaluation paths, whose union forms an execution graph [4]. Each transition in the graph corresponds to the evaluation of *one* single point of a data field. Rather than considering data fields at a programming point of view, we thus propose a more abstract approach, using *variables*, i.e. instances of a data field. This enables us to manipulate homogeneous sets.

We denote by $\mathcal{X}$ the set of variables. We denote by *Out* the set of output variables, and by *In* the set of input variables with a large sense: these variables are the usual input variables, together with those variables whose evaluation expression is a constant one. We have $In \cup Out \subseteq \mathcal{X}$. The set of values is denoted by $\mathcal{V}$. An additional value $\perp$ denotes undefinedness.

An equation associates an evaluation expression to a variable. It is written $x = e(y_1, \ldots, y_n)$, or $x = e(\vec{y})$. By construction, there is exactly one defining equation $x = e(y_1, \ldots, y_n)$, $n \geq 1$ (or $x = e(\vec{y})$ for short), for each variable $x \in \mathcal{X} \backslash In$. As a convention, the equation corresponding to x is denoted by E_x.

A program is a set of equations: $S \equiv \prod_{x \in \mathcal{X} \backslash In} E_x$. Note that it is a simplification of the usual notion of an ALPHA program, since we don't consider here the declarations of input, local or output variables, implicitely contained in the definition of *In* and *Out*.

An environment σ is an application from the set of variables $\mathcal{X}$ to the set of values $\mathcal{V} \cup \perp$. The set of environments is denoted by Σ. The support of some environment σ is the set of variables that are valued in σ: $supp(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq \perp\}$. We distinguish a particular subset of Σ, that is the set of *initial* environments denoted by *Init*: $\sigma \in Init$ if and only if $supp(\sigma) = In$. As usual, environments are generalized to expressions and yield strict applications from expressions to $\mathcal{V} \cup \perp$.

We denote by $\preceq$ the reflexive, transitive closure of the dependance relation induced by the program's equations: $y \preceq x$ if and only if there exists a sequence $(x_0, \ldots, x_n)$ such that $x_0 = y$, $x_n = x$, and for all $1 \leq i \leq n$, $E_{x_i} \equiv x_i = e(\ldots, x_{i-1}, \ldots)$. Such a sequence is called a *dependence path*.

Two approaches have been proposed in [4] as a basis for an operational semantics of ALPHA programs. They differ in the way the "execution graph" is generated. The first one is the dataflow approach: an expression is evaluated as soon as its free variables have been given a value. The second one is a demand-driven approach: an expression is evaluated when its free variables are known, but only if this evaluation is necessary to the computation of the output variables.

2.1 The dataflow approach

In this approach, we consider that a programs produces its output values starting from the input. The most important thing we have to know about an equation is wether its expression has all its free variables already valued. We give the following definition.

Definition 4 (Readiness) *We say that equation $E_x \equiv x = e(\vec{y})$ is* ready in σ *iff* $\forall i, \sigma(y_i) \neq \bot \wedge \sigma(x) = \bot$.

We can now give the formal definition of the transition function for the dataflow approach.

Transition function

Rule 1

$$\frac{x = e(\vec{y}) \text{ ready in } \sigma}{\sigma \xrightarrow{x} \sigma[x \leftarrow \sigma(e(\vec{y}))]}$$

A sequence $(\sigma_i)_{i \in \{0,\ldots,n\}}$ of environments such that $\forall i \in \{1, \ldots, n\}, \sigma_{i-1} \xrightarrow{x_i} \sigma_i$ is called a transition sequence.

2.2 The demand-driven approach

We don't consider any more that an ALPHA program must "consume" its input variables, but that it must "produce" its output variables. In that sense, we restrict computations to those which are necessary for the evaluation of output variables.

Considering a subset V of Out, the set of *necessary* variables w.r.t. V is the set of variables appearing on a path between In and V. This is formalized in the following definition.

Definition 5 (Necessary) *Let V be a subset of Out. The variables that are* necessary *to V are the variables appearing on a path between In and V.*

$$Necessary(V) = \{x \mid \exists z \in In, \exists t \in V,\ z \preceq x \preceq t\}$$

As a convention. $Necessary(Out)$ will simply be denoted by $Necessary$.

Transition function

This second rule is defined with respect to some subset V of Out.

Rule 2

$$\frac{x = e(\vec{y}) \text{ ready in } \sigma \wedge x \in Necessary(V)}{\sigma \underset{V}{\overset{x}{\Longrightarrow}} \sigma[x \leftarrow \sigma(e(\vec{y}))]}$$

In the case where $V = Out$, we simply write $\sigma \overset{x}{\Longrightarrow} \sigma'$.

2.3 Equivalence between the two operational semantics

As the second rule is a restriction of the first one, the two operational semantics cannot be strictly equivalent. We thus will define a notion of equivalence *modulo* some set of (output) variables.

Definition 6 (Equivalence modulo a set of variables) *Let V be a set of variables, and σ,σ' two environments. We say that σ and σ' are* equivalent modulo V, $\sigma \underset{V}{\sim} \sigma'$ *iff*

$$V \subseteq supp(\sigma) \qquad V \subseteq supp(\sigma') \qquad \forall x \in V,\ \sigma(x) = \sigma'(x).$$

We now show that, from every sequence of transitions obtained by means of the first transition rule, we are able to obtain a *rearranged* sequence where all variables of *Necessary* are valued first. More precisely, we show the following proposition.

Proposition 1 (Rearrangement) *Let V be a finite subset of Out, and σ_0 an environment in Init. If there exists a sequence*

$$\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-1}} \sigma_n$$

with $V \subseteq supp(\sigma_n)$, then there exists a sequence

$$\sigma_0 = \sigma'_0 \underset{V}{\overset{x'_0}{\Longrightarrow}} \sigma'_1 \underset{V}{\overset{x'_1}{\Longrightarrow}} \ldots \underset{V}{\overset{x'_{k-1}}{\Longrightarrow}} \sigma'_k \xrightarrow{x'_k} \sigma'_{k+1} \xrightarrow{x'_{k+1}} \ldots \xrightarrow{x'_{n-1}} \sigma'_n = \sigma_n$$

such that

$$\begin{cases} (x'_0, \ldots, x'_{n-1}) \text{ is a permutation of } (x_0, \ldots, x_{n-1}) \\ \forall j \in \{k, \ldots, n-1\},\ x'_j \notin Necessary(V) \\ \sigma'_k \underset{V}{\sim} \sigma_n \end{cases}$$

The proof of this proposition requires a preliminary lemma. This lemma has been informally proved in [4]. It expresses that the value of a variable does not change once it has been computed and is different from $\perp$.

Lemma 1 (Value Preservation) *Let S be an* ALPHA *program , $\sigma_0 \to \sigma_1 \to \ldots \to \sigma_k$ a sequence of transitions for this program, x a variable of S. If $\sigma_0(x) \neq \perp$, then for all $0 \leq j \leq k$, we have $\sigma_j(x) = \sigma_0(x)$.*

Proof *We show by induction on j in $\{0, \ldots, k\}$ that we have $\sigma_j(x) = \sigma_0(x)$.*

It is trivially true for $j = 0$.

Let us assume that this property is established for j, with $j < k$. By induction hypothesis, we have $\sigma_j(x) = \sigma_0(x)$, thus $\sigma_j(x) \neq \perp$. Transition $\sigma_j \to \sigma_{j+1}$ thus corresponds to the valuation of a variable x_j, with $x_j \neq x$: $\sigma_{j+1} = \sigma_j[x_j \leftarrow \sigma_j(e(\vec{y_j}))]$. As computations do not induce any side effects, we have $\sigma_{j+1}(x) = \sigma_0(x)$. The property is thus established for $j+1$. ❑

We now can give the proof of Proposition 1.

Proof

The proof is done by induction on the length of the transition sequence. The result is trivial for a sequence of length 1. Let us assume that we have established it for any sequence of length $n-1$, and let us consider a sequence $\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-1}} \sigma_n$ matching the hypothesis. We distinguish between two cases depending on x_{n-1}.

- *If $x_{n-1} \notin Necessary(V)$. We apply the induction hypothesis to $\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-2}} \sigma_{n-1}$. We thus get a sequence*

$$\sigma_0 = \sigma'_0 \underset{V}{\overset{x'_0}{\Longrightarrow}} \sigma'_1 \underset{V}{\overset{x'_1}{\Longrightarrow}} \ldots \underset{V}{\overset{x'_{k-1}}{\Longrightarrow}} \sigma'_k \xrightarrow{x'_k} \sigma'_{k+1} \xrightarrow{x'_{k+1}} \ldots \xrightarrow{x'_{n-2}} \sigma'_{n-1} = \sigma_{n-1}$$

where $(x'_0, \ldots, x'_{n-2})$ is a permutation of $(x_0, \ldots, x_{n-2})$ and $\sigma'_k \underset{V}{\sim} \sigma_{n-1}$. As $x_{n-1} \notin Necessary(V)$ and $V \subseteq Necessary(V)$, we have $x_{n-1} \notin V$, so $\sigma'_k \underset{V}{\sim} \sigma_n$. Adding transition $\sigma_{n-1} \xrightarrow{x_{n-1}} \sigma_n$ to the previous transition sequence thus yields the desired sequence.

- *If $x_{n-1} \in Necessary(V)$. We first apply the induction hypothesis to $\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-2}} \sigma_{n-1}$. We thus get a sequence*

$$\sigma_0 = \sigma'_0 \underset{V}{\overset{x'_0}{\Longrightarrow}} \sigma'_1 \underset{V}{\overset{x'_1}{\Longrightarrow}} \ldots \underset{V}{\overset{x'_{k-1}}{\Longrightarrow}} \sigma'_k \xrightarrow{x'_k} \sigma'_{k+1} \xrightarrow{x'_{k+1}} \ldots \xrightarrow{x'_{n-2}} \sigma'_{n-1} = \sigma_{n-1}$$

where $(x'_0, \ldots, x'_{n-2})$ is a permutation of $(x_0, \ldots, x_{n-2})$ and $\sigma'_k \underset{V}{\sim} \sigma_{n-1}$. We now have two cases to consider.

  - *If $k = n-1$. Taking $\sigma'_0 \underset{V}{\overset{x'_0}{\Longrightarrow}} \sigma'_1 \underset{V}{\overset{x'_1}{\Longrightarrow}} \ldots \underset{V}{\overset{x'_{n-2}}{\Longrightarrow}} \sigma'_{n-1} \underset{V}{\overset{x_{n-1}}{\Longrightarrow}} \sigma_n$ yields the desired sequence.*
  - *If $k < n-1$. We have $x_{n-2} \notin Necessary(V)$. Let us prove that we do not have $x_{n-2} \preceq x_{n-1}$. As in the initial transition sequence x_{n-2} has been valued in a sequence starting from an initial state, x_{n-2} is not in In and the set $\{y \mid y \preceq x_{n-2}\}$ is finite. Let us consider a minimal element for $\preceq$ in this nonempty set. We claim that this element is a variable of In. Otherwise it would depend on some other variable and would not be minimal. We thus have some x in In such that $x \preceq x_{n-2}$. If we had $x_{n-2} \preceq x_{n-1}$, we would have $x \preceq x_{n-2} \preceq x_{n-1}$. As $x_{n-1} \in Necessary(V)$, we would also have $x_{n-2} \in Necessary(V)$. That is impossible, and we conclude that $x_{n-2} \not\preceq x_{n-1}$.*
    We are now able to "permute" x_{n-1} and x_{n-2}. We know that the equation defining x_{n-1} is ready in σ'_{n-1}. As $x_{n-2} \not\preceq x_{n-1}$, this equation is also ready in σ'_{n-2}. We thus can construct the following transition sequence:

$$\sigma'_0 \underset{V}{\overset{x'_0}{\Longrightarrow}} \sigma'_1 \underset{V}{\overset{x'_1}{\Longrightarrow}} \ldots \underset{V}{\overset{x'_{k-1}}{\Longrightarrow}} \sigma'_k \xrightarrow{x'_k} \sigma'_{k+1} \xrightarrow{x'_{k+1}} \ldots \sigma'_{n-2} \underset{V}{\overset{x'_{n-1}}{\Longrightarrow}} \sigma''_{n-1} \xrightarrow{x'_{n-2}} \sigma''_n.$$

    Moreover, Lemma 1 guaranties that

$$\sigma''_n = \sigma_n.$$

    We are brought back to the first case, where we had $x_{n-1} \notin Necessary(V)$. We can apply the induction hypothesis once more to get the desired result.

Proof of Proposition 1 is done. ❑

Proposition 1 is the key to establish the following equivalence theorem between the two operational approaches.

Theorem 1 (Equivalence) *Let V be a finite subset of Out, and σ_0 an environment in $Init$. If there exists a sequence*

$$\sigma_0 \xrightarrow{x_0} \sigma_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{n-1}} \sigma_n$$

with $V \subseteq supp(\sigma_n)$, then there exists a sequence

$$\sigma_0 \underset{V}{\overset{x'_0}{\Longrightarrow}} \sigma'_1 \underset{V}{\overset{x'_1}{\Longrightarrow}} \ldots \underset{V}{\overset{x'_{m-1}}{\Longrightarrow}} \sigma'_m$$

with $\sigma_n \underset{V}{\sim} \sigma'_m$ and $\{x'_0, \ldots, x'_m\} \subseteq \{x_0, \ldots, x_n\}$.

The converse is also true.

Proof *The proof is very simple. For the first part of the theorem, it is just a consequence of Proposition 1, since the beginning of the rearranged sequence is only made of variables taken from $Necessary(V)$.*

For the converse part, we just have to remark that Rule 2 is a restriction of Rule 1. We thus can take the same sequence.

❑

2.4 Confluence properties

We finally give some properties of the operational semantics. These properties express the fact that, although the computations of an ALPHA program are made in a non-deterministic manner, the result of these computations is deterministic.

Proposition 2 *Let $\sigma_0 \longrightarrow^* \sigma$ be a transition sequence, and let x be a variable of Out. If $\sigma(x) \neq \bot$, then $\forall y \in Necessary(\{x\}) : \sigma(y) \neq \bot$.*

Proof *The proof is straightforward, by definition of the transition rule and Lemma 1.* ❑

Proposition 3 *Let V be a finite subset of Out. Let σ_0 be in Init and $\sigma_0 \underset{V}{\Longrightarrow}^* \sigma$ a transition sequence such that $supp(\sigma) = V$. Then there does not exist any transition $\sigma \underset{V}{\Longrightarrow}$.*

Proof *This is a direct consequence of Proposition 2: as $supp(\sigma) = V$, we have $\sigma(x) \neq \bot$ for all x in V, so $\sigma(y) \neq \bot$ for all y in $Necessary(V)$.* ❑

Proposition 4 (Confluence) *Let V be a finite subset of Out. If $\sigma_0 \underset{V}{\Longrightarrow}^* \sigma_1$ and $\sigma_0 \underset{V}{\Longrightarrow}^* \sigma_2$, then there exists some σ' such that $\sigma_1 \underset{V}{\Longrightarrow}^* \sigma'$ and $\sigma_2 \underset{V}{\Longrightarrow}^* \sigma'$.*

Proof *Let $\sigma \overset{V_1}{\underset{V}{\Longrightarrow}}^* \sigma_1$ and $\sigma \overset{V_2}{\underset{V}{\Longrightarrow}}^* \sigma_2$ be two transition sequences, where $V_1 = supp(\sigma_1) \setminus supp(\sigma)$ and $V_2 = supp(\sigma_1) \setminus supp(\sigma)$. We have $V_1 \cup V_2 \subseteq Necessary(V)$*

Let $\sigma_1 \overset{V_2 \setminus V_1}{\underset{V}{\Longrightarrow}}^* \sigma_1'$ be the transition sequence such that the variables of $V_2 \setminus V_1$ appear in the same order as in $\sigma \underset{V}{\Longrightarrow}^* \sigma_2$. As we have the same order, and as $V_1 \cap V_2 \subseteq supp(\sigma_1)$, this is a valid transition sequence. Moreover, we have $supp(\sigma_1') = V_1 \cup V_2$.

Let now $\sigma_2 \overset{V_1 \setminus V_2}{\underset{V}{\Longrightarrow}}^* \sigma_2'$ be the transition sequence such that the variables of $V_1 \setminus V_2$ appear in the same order as in $\sigma \underset{V}{\Longrightarrow}^* \sigma_1$. We have $supp(\sigma_2') = V_1 \cup V_2$.

From Lemma 1, we get $\sigma' = \sigma_1' = \sigma_2'$. ❑

Proposition 5 (Determinism) *Let V be a finite subset of Out. If*

$$\sigma_0 \underset{V}{\Longrightarrow}^* \sigma_1$$
$$\sigma_0 \underset{V}{\Longrightarrow}^* \sigma_2$$
$$supp(\sigma_1) = supp(\sigma_2) = V,$$

then

$$\sigma_1 = \sigma_2$$

Proof *This is a straightforward consequence of Propositions 3 and 4.* ❑

3 Proving ALPHA programs

In this section, we present a proof system for ALPHA. We give a general definition on the entire language, but some interesting properties will be established only on a subset of ALPHA. This subset is composed of programs for which we are able to give a parametrization of the set of variables into finite subsets.

3.1 Assertion language

We want to define a language that enables us to express properties about the values of the program variables.

- An index expression is an integer constant, an index variable, or the combination of index expressions with usual arithmetical operators (including modulo).
- An arithmetic expression is an integer constant, an integer ALPHA data field indexed by a tuple of index expressions, or the combination of arithmetic expressions with usual operators. The size of the tuple indexing a data field corresponds to the dimension of this data field.
- A boolean expression is a boolean constant, a boolean data field indexed by a tuple of index expressions, or the combination of arithmetic or index expressions with usual relation operators ($=, <, \ldots$).

- A formula of the assertion language is the combination of boolean expressions with usual boolean connectives ($\wedge, \vee, \neg, \Rightarrow$), or a quantified formula. The usual existential and universal quantifiers apply on indices in the domain of a data field. For instance, $\forall t, p : X[t, p] = 0$ means that X is equal to zero on its entire domain. Formulae must be closed, and we assume implicit external quantification.

The evaluation of a formula in a given environment has to take into account the use of quantification of indices. We thus introduce a new kind of environments: (σ, ρ) is a pair made from a program environment σ and an index environment ρ, i.e. a mapping from index names to integer values. For instance, if A is a one-dimensional data field, the value of variable $A[t]$ in (σ, ρ) is $\sigma(A[\rho(t)])$. The special index environment ρ_0 is the empty index environment, where all index values are undefined. The validity of a closed formula P in an environment σ is denoted by $\sigma \models P$. It corresponds to the validity of P in (σ, ρ_0) and it is defined in the following way.

- The interpretation of arithmetic or relation operators is standard, but we assume the following: **if a variable is undefined in some (σ, ρ), then the interpretation of any boolean expression involving this variable will be** *true*. Within this special interpretation, it is possible to express that a given variable is defined: for instance, expression $(X[i] \neq X[i])$ evaluates to true in σ if and only if $\sigma(X[i]) = \perp$, that is, $X[i]$ is undefined in σ.
- We keep the usual meaning to the usual boolean connectives. For instance, $(\sigma, \rho) \models (P \Rightarrow Q)$ iff $(\sigma, \rho) \models P$ implies $(\sigma, \rho) \models Q$.
- For the quantifiers, we have
  - $(\sigma, \rho) \models \exists i : P$ iff there exists some integer v such that $(\sigma, \rho[i \leftarrow v]) \models P$.
  - $(\sigma, \rho) \models \forall i : P$ iff for all integers v, we have $(\sigma, \rho[i \leftarrow v] \models P$.

3.2 Specifications and validity

We aim at giving ALPHA an assertional semantics inspired from the Hoare logic for imperative languages. In classical Hoare logic, we define formulae like $\{P\}\ S\ \{Q\}$, where S is a program and P, Q are two assertions. Such a formula means that, if we start from a state satisfying precondition P, execution of S yields a state satisfying postcondition Q. This gives birth to two notions of correctness of a specification formula, respectively total or partial correctness, depending on wether we consider that S must terminate or not.

In the case of ALPHA programs, we would like to write formulae like $\{P\}\ S\ \{Q\}$, which would mean "If P is true in some initial state, then Q is true in some final state after execution of S." The notion of initial state is not so difficult to define: we just have to take environments $\sigma \in \mathit{Init}$. But, because of the "dataflow" nature of the language, an ALPHA program may not terminate, even if it yields the expected results. As a consequence, the notion of terminating program and of final state are not easy to translate. As for the "demand-driven" operational semantics, we restrict ourselves to some finite subsets of the program's variables.

We thus can make the notion of validity (or correctness) of a specification precise. Validity is defined with respect to a finite set of output variables.

Definition 7 (Validity on a finite subset) *Let $\{P\}\ S\ \{Q\}$ be a specification formula, and V a finite subset of Out. We say that this specification is* valid w.r.t. *V, which is denoted by*

$$\models_V \{P\}\ S\ \{Q\},$$

if for any $\sigma \in \mathit{Init}$ such that $\sigma \models P$, for any sequence $\sigma \longrightarrow^* \sigma'$ such that $V \subseteq \mathit{supp}(\sigma')$, we have

$$\sigma' \models Q.$$

We can give a more general definition of validity, generalizing the first one.

Definition 8 (Validity) *Let $\{P\}\ S\ \{Q\}$ a specification formula. We say that this specification is* valid, *which is denoted by*

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

if for any V, finite subset of Out, we have

$$\models_V \{P\}\ S\ \{Q\}.$$

Validity has been defined with respect to the "dataflow" operational semantics. But if we consider only a particular kind of postconditions, we can take into account the demand-driven operational semantics.

Definition 9 (Stability) *Let Q be a predicate. We say that Q is* stable *on some subset V if*

$$\forall \sigma, \sigma' : (\sigma \underset{V}{\sim} \sigma' \Rightarrow (\sigma \models Q \text{ iff } \sigma' \models Q)).$$

Proposition 6 (Demand-driven validity) *If Q is stable on V and*

$$\left.\begin{array}{l} \sigma \in Init \\ \sigma \models P \\ \sigma \underset{V}{\Longrightarrow}^* \sigma' \\ supp(\sigma') \cap Out = V \end{array}\right\} \quad \Rightarrow \quad (\sigma' \models Q),$$

then

$$\models_V \{P\}\ S\ \{Q\}.$$

Proof *This is a direct consequence of Proposition 1 and of the definition of a stable predicate.* ❑

3.3 Invariants and provability

We now want to introduce the notion of *provability* for a specification. Due to the iterative and non-deterministic nature of the operational semantics, the proof of a specification is established by producing an invariant. The notion of invariant defined here is relative to a program, to a subset of initial states and to a subset of output variables.

Definition 10 (Invariant) *A formula I is an* invariant *for the program S, the set $\mathcal{I} \subseteq Init$ and the finite set $V \subseteq Out$ iff*

$$\text{for all } \sigma \text{ such that } \exists\ \sigma_0 \in \mathcal{I} : \sigma_0 \underset{V}{\Longrightarrow}^* \sigma, \quad \sigma \models I$$

where $\underset{V}{\Longrightarrow}$ denotes a transition relative to program S and subset V.

Finally, we can give the definition of provability. As for validity, provability is first defined on a finite subset, and extended thereafter.

Definition 11 (Provability) *Specification $\{P\}\ S\ \{Q\}$ is said to be* provable *w.r.t. a finite subset V of Out, which is denoted by*

$$\vdash_V \{P\}\ S\ \{Q\},$$

if there exists an invariant I relative to S, to $\mathcal{I}_P = \{\sigma \in Init \mid \sigma \models P\}$, and to V such that

$$\text{for all } \sigma \text{ such that } supp(\sigma) \cap Out = V, \qquad (\sigma \models I) \Rightarrow (\sigma \models Q).$$

More generally, specification $\{P\}\ S\ \{Q\}$ is said to be provable*, which is denoted by*

$$\vdash \{P\}\ S\ \{Q\},$$

if for any finite subset V of Out, we have

$$\vdash_V \{P\}\ S\ \{Q\}.$$

3.4 Example

We give an example of specification and invariant for the convolution program of Section 1. We first recall the program.

```
system convolution( a : { j | 1<=j<=4 } of integer;
                    x : { i | i>=1 } of integer )
      returns ( y : { i | i>=4 } of integer;
   var
      Y : { i,j | 0<=j<=4 ; i>=4 } of integer;
   let
      Y[i,j] = case
               {| j=0 } : 0;
               {| 1<=j<=4} : Y[i,j-1] + a[j]*x[i-j+1];
               esac;
      y[i]=Y[i,4];
   tel
```

In this example, we have

$$\begin{aligned} In &= \{a[j] \mid 1 \leq j \leq 4\} \cup \{x[i] \mid 1 \leq i\} \cup \{Y[i,0] \mid 1 \leq i\} \\ Out &= \{y[i] \mid 1 \leq i\} \end{aligned}$$

Let us fix a value for i, being greater or equal than 1, and let $|.|$ denote the absolute value function. A specification could be

$$\begin{aligned} P &\equiv \forall j : \ (1 \leq j \leq 4) \Rightarrow |a[j]| \leq M \\ &\qquad \wedge |x[i+j-1]| \leq K \\ Q &\equiv |y[i]| \leq 4KM \end{aligned}$$

Taking $V = \{y[i]\}$, we have trivially

$$\models_V \{P\}\ S\ \{Q\}.$$

There are several invariants. For instance, a very simple one is

$$|y[i]| \leq 4KM,$$

but it is not of much use, since its variable is defined only in a final state. There is therefore no hope to use it in a formal proof. It is very much like selecting *true* as the invariant of a while loop in proving a Pascal program!

An other one, more valuable, is

$$\begin{aligned} & \forall j : (1 \leq j \leq 4) \Rightarrow (|Y[i,j]| \leq jKM) \\ \wedge \quad & y[i] = Y[i,4] \\ \wedge \quad & \Delta(y[i]) \Rightarrow \Delta(Y[i,4]) \end{aligned}$$

where $\Delta(x)$ is a shorthand for an expression meaning "x is defined"[1] (see Section 3.1). We will show below it is in some sense its *canonical* weakest invariant.

3.5 Soundness

A crucial point now is to check that our definition of provability is valid (or *sound*), that is anya provable specification is correct w.r.t. the operational semantics. This problem is adressed by the following theorem.

Theorem 2 (Soundness) *Let V be a finite subset of Out, and $\{P\}\ S\ \{Q\}$ be a specification such that Q is stable on V. If*

$$\vdash_V \{P\}\ S\ \{Q\},$$

then

$$\models_V \{P\}\ S\ \{Q\}.$$

Proof *Let $\mathcal{I}_P$ be the set $\{\sigma \in Init \mid \sigma \models P\}$. By definition of provability, there exists a formula I such that*

[1] Note that $\Delta(x)$ itself is not a formula, but just a notation.

i. for all σ such that $\exists \sigma_0 \in \mathcal{I}_P : \sigma_0 \underset{V}{\Longrightarrow}^* \sigma, \quad \sigma \models I$;

ii. for all σ such that $V = supp(\sigma) \cap Out$, $(\sigma \models I) \Rightarrow (\sigma \models Q)$.

Let us take a sequence $\sigma \underset{V}{\Longrightarrow}^* \sigma'$ such that $\sigma \in Init$, $\sigma \models P$ and $supp(\sigma') \cap Out = V$. We thus have $\sigma' \models I$. Moreover, by (ii) we have $\sigma' \models I \Rightarrow Q$. We have proved that

$$\sigma' \models Q.$$

As Q is stable on V, Proposition 6 yields

$$\models_V \{P\}\ S\ \{Q\}.$$

□

3.6 Unambiguous invariants

In section 3.3, we have given a general definition of an invariant. We now distinguish a special class of invariants, i.e. the class of *unambiguous* invariants. These invariants will be helpful in the next sections to establish some properties such as completeness of the proof system or equivalence between ALPHA programs.

We first give a general definition of an unambiguous assertion.

Definition 12 (Unambiguous assertion) *Let P be an assertion. We say that P is* unambiguous *with respect to some finite subset V of Out if, for any pair of environments σ and σ', we have*

$$\left.\begin{array}{c} \sigma \models P \\ \sigma' \models P \\ V \subseteq supp(\sigma) \\ V \subseteq supp(\sigma') \\ \sigma \underset{In}{\sim} \sigma' \end{array}\right\} \Rightarrow \quad \sigma \underset{V}{\sim} \sigma'$$

This definition applies to any assertion, but we essentially use it for invariants. Intuitively, an invariant is unambiguous if it catches all the semantics of the program. In other words, a computation satisfying the invariant always yields the same single result on the considered subset.

3.7 Restricted completeness

In section 3.5, we have seen that our notion of provability was sound, i.e. that every provable specification is valid. We now want to adress the converse problem: is any valid specification provable? In other words, are we always able to produce an invariant for a given valid specification?

In the case of imperative languages, proving completeness of a proof system is quite complex. The proof essentially relies on the definition of a weakest preconditions calculus and on the encoding of invariants in the assertion language. Complexity comes from this encoding: transformations of environments must be translated into arithmetic manipulations. In the case we consider, due to the particular form of the language based on explicit defining equations, the expression of invariants is much simpler.

The property of completeness that we prove here is not fully general. It only concerns a certain class of programs, depending on a property of the dependence graph. We give the following definition.

Definition 13 (Finitely closed) *Let G be the dependence graph of an* ALPHA *program. We say that G is finitely (left-) closed if, for every finite subset V of G, there exists a finite subset W of G that contains V and that is left-closed for the dependence relation:*

$$\forall x \in W : \forall y \in G : ((y \preceq x) \Rightarrow (y \in W))$$

By extension, a program is finitely closed iff its dependence graph is finitely left-closed.

Proving completeness only for finitely closed programs is not too restrictive, since this class contains all "useful" programs, i.e. programs that have computable results. We give examples in the next section.

We now just have to define a particular kind of invariants: as shown in the following definition, a canonical invariant simply expresses that, as soon as a variable is defined, its value is equal to the value of its definition

expression. This definition uses some additional notations and shorthands: χ denotes the set of all non-input data fields in the program; $\vec{\imath} \in \mathcal{D}_X$ is a shorthand for the set of affine constraints defining the domain of X; $\Delta(X[\vec{\imath}])$ is a shorthand for the formula $\neg(X[\vec{\imath}] \neq X[\vec{\imath}])$ meaning "$X[\vec{\imath}]$ is defined" (see Section 3.1), and $\Delta(\vec{Y}[\vec{f}(\vec{\imath})])$ is a shorthand for $\bigwedge_{Y \in \vec{Y}} \Delta(Y[f(\vec{\imath})])$.

Definition 14 (Canonical invariant) *Let S be a program and P an assertion.*

$$\prod_{X \in \chi} \forall \vec{\imath} \in \mathcal{D}_X : (X[\vec{\imath}] = e(\vec{Y}[\vec{f}(\vec{\imath})]))$$

The canonical invariant *I_C of the pair (P, S) is defined as*

$$\begin{array}{rcll} I_C & \equiv & \bigwedge_{X \in \chi} \forall \vec{\imath} \in \mathcal{D}_X : & (X[\vec{\imath}] = e(\vec{Y}[\vec{f}(\vec{\imath})])) \\ & & & \wedge\, \Delta(X[\vec{\imath}]) \Rightarrow \Delta(\vec{Y}[\vec{f}(\vec{\imath})]) \\ & & \wedge\, P & \end{array}$$

From this definition, we immediately deduce the following proposition.

Proposition 7 *Let S be a finitely closed* Alpha *program and P an assertion. For any finite subset V of Out, the canonical invariant of (P, S) is unambiguous w.r.t. V.*

Proof *Let I_C be the canonical invariant for (P, S), and let us suppose that I_C is not unambiguous. There exist two environments σ and σ' such that*

$$\begin{array}{l} \sigma \models I_C \\ \sigma' \models I_C \\ V \subseteq supp(\sigma) \\ V \subseteq supp(\sigma') \\ \sigma \underset{In}{\sim} \sigma' \end{array} \quad \text{and} \quad \sigma \underset{V}{\not\sim} \sigma'$$

Let us first remark that if $\sigma \models I_C$ and $V \subseteq supp(\sigma)$, then $\forall x \in Necessary(V)$, we have $\sigma(x) \neq \bot$ and $\sigma(x) = \sigma(e(\vec{y}))$. Thus there exists some variable x in V such that $\sigma(x) \neq \sigma'(x)$, $\sigma(x) \neq \bot$ and $\sigma'(x) \neq \bot$. As S is finitely closed, all paths leading to x in the dependence graph are of finite length. For any variable $z \notin In$ such that $\sigma(z) \neq \sigma'(z)$, there exists some $y \preceq z$ such that $\sigma(y) \neq \sigma'(y)$. We deduce that there exists some variable y of In, starting point of a path leading to x, such that $\sigma(y) \neq \sigma'(y)$, which is impossible. ❑

After all these preliminary works, we can give the completeness theorem.

Theorem 3 (Restricted completeness) *Let S be a finitely closed* Alpha *program, $\{P\}\ S\ \{Q\}$ a specification for this program, and V a finite subset of Out. If*

$$\models_V \{P\}\ S\ \{Q\},$$

then

$$\vdash_V \{P\}\ S\ \{Q\}.$$

Proof *Let us assume that $\models_V \{P\}\ S\ \{Q\}$, and let I_C be the canonical invariant of (P, S). We must prove*

i. for all σ such that $\exists \sigma_0 \in Init, \sigma_0 \models P$ and $\sigma_0 \underset{V}{\Longrightarrow}^* \sigma$, $\sigma \models I_C$;

ii. for all σ such that $V = supp(\sigma) \cap Out$: if $\sigma \models I$, then $\sigma \models Q$.

The first item is proved by induction on the length n of the transition sequence $\sigma_0 \underset{V}{\Longrightarrow}^n \sigma$. If $n = 0$, we have $\sigma = \sigma_0$. As $\sigma_0 \models P$ and σ is undefined on non-input variables, we have $\sigma \models I_C$. Let us now assume the result established for $n-1$ and let us consider a sequence $\sigma_0 \underset{V}{\Longrightarrow}^n \sigma$. There exist some variable x and some state σ' such that $\sigma_0 \underset{V}{\Longrightarrow}^{n-1} \sigma' \underset{V}{\overset{x}{\Longrightarrow}} \sigma$. By induction hypothesis, we know that $\sigma' \models I_C$. The last

transition is valid, so $\sigma'(x) = \bot$ *and* $\sigma(x) = \sigma'(e(\vec{y}))$. *Variable* x *corresponds to some data field* X *and some index* $\vec{\imath}$. *In* σ *we have* $\Delta(X[\vec{\imath}]) = true$. *As* $\sigma(x) = \sigma'(e(\vec{y}))$, *we have* $\sigma \models X[\vec{\imath}] = e(\vec{Y}[\vec{f}(\vec{\imath})])$. *In* σ' *we have* $\Delta(\vec{Y}[\vec{f}(\vec{\imath})]) = true$, *so it remains true in* σ. *Moreover,* $\Delta(X[\vec{\imath}])$ *has become true in* σ: *we conclude that*

$$\sigma \models \Delta(X[\vec{\imath}]) \Rightarrow \Delta(\vec{Y}[\vec{f}(\vec{\imath})]).$$

Finally, as P *involves input variables only, it is left unchanged by the transition.*

Let us prove the second item. Let us consider some σ *such that* $V = supp(\sigma) \cap Out$ *and* $\sigma \models I_C$. *Let* σ_1 *be the restriction of* σ *to* In, *which is denoted by* $\sigma_1 = \sigma|_{In}$: *we have* $supp(\sigma_1) = In$ *and* $\sigma_1 \underset{In}{\sim} \sigma$. *As* $\sigma \models I_C$, *we know that* $\sigma_1 \models P$. *Let now* σ_1' *be such that* $V \subseteq supp(\sigma_1')$ *and*

$$\sigma_1 \underset{V}{\Longrightarrow}^* \sigma_1'.$$

Such a σ_1' *exists because* S *is finitely closed. Moreover,* σ_1' *satisfies the canonical invariant. As this invariant is unambiguous, we conclude that* $\sigma_1'|_V = \sigma'|_V$. *Finally, since specification* $\{P\}\ S\ \{Q\}$ *is valid,* $\sigma_1' \models Q$, *so*

$$\sigma' \models Q.$$

❑

4 Finite parametrization of ALPHA programs

In Section 3.7, we have restricted ourselves to a special class of programs, i.e. finitely closed programs. We now give a more intuitive presentation of this class. Any finitely closed program can in fact be seen as an enumeration of "slices", i.e. finite closed subsets of variables. We just have to remark that we are able to give an enumeration of all finite subsets of *Out* because this set is enumerable, and that any finite subset is "covered" by a finite number of these enumerated subsets. As expressed by the following definition, such an enumeration of finite subsets is called a *parametrization*.

Definition 15 (Parametrization) *Let* G *be the dependence graph of an* ALPHA *program. A* parametrization *of* G *is a function* φ *such that*

$$\begin{array}{rll} & \varphi: & \mathbb{N} \longrightarrow \mathcal{P}(G) \\ & & t \mapsto \varphi(t) \\ \\ \textit{with} & & \forall t \in \mathbb{N} : \varphi(t) \textit{ is finite} \\ & \wedge & \bigcup_{t \in \mathbb{N}} \varphi(t) = G \\ & \wedge & \forall t \in \mathbb{N} : \forall x \in \varphi(t) : \forall y \in G : ((y \preceq x) \Rightarrow y \in \varphi(t)) \quad (*) \end{array}$$

A parametrization of a program is a parametrization of its dependence graph.

A parametrization is a particular ordering of finite subsets of variables for a given finitely closed program.

Proposition 8 *Any finitely closed program can be given a parametrization, and conversely.*

Proof *Let us assume that there exists some* φ *respecting the hypothesis, and let us fix some finite subset* V. *There exists some finite set* $\{t_1, \ldots, t_n\}$ *such that* $V \subseteq \varphi(t_1) \cup \ldots \cup \varphi(t_n)$. *All* $\varphi(t_i)$ *are finite and left-closed. To find* W *matching the hypothesis of Definition 13, we just have to take* $W = \varphi(t_1) \cup \ldots \cup \varphi(t_n)$.

Conversely, there exists an enumeration of all finite subsets $V_1, \ldots, V_n, \ldots$ *Let* W_i *be the left-closed set corresponding to* V_i. *Thus, taking* $\varphi(t) = W_t$ *yields a valid (possibly redundant) parametrization.*

❑

Examples

The first example is the convolution filter, with this time a parameter N giving the width of the convolution.

```
system convolution( N : { N | N>=0 } parameter;
                    a : { j | 1<=j<=N } of integer;
                    x : { i | i>=1 } of integer )
       returns ( y : { i | i>=N } of integer;
    var
       Y : { i,j | 0<=j<=N ; i>=N } of integer;
    let
       Y[i,j] = case
                {| j=0 } : 0;
                {| 1<=j<=N} : Y[i,j-1] + a[j]*x[i-j+1];
                esac;
       y[i]=Y[i,N];
    tel
```

This program is obviously finitely closed. Let us first consider that N has a fixed value. By defining $t = i-1$ for $i \geq 1$, we can take

$$\begin{array}{rcl} \varphi(t) & = & \{x_j \mid i \leq j \leq i+N-1\} \\ & \cup & \{a_j \mid 1 \leq j \leq N\} \\ & \cup & \{Y_j \mid 0 \leq j \leq N\} \\ & \cup & \{y_i\} \end{array}$$

as illustrated on Fig 4. Circled dots are vertices of $\varphi(t)$ for $t = i-1 = 9$ and for $N = 6$.

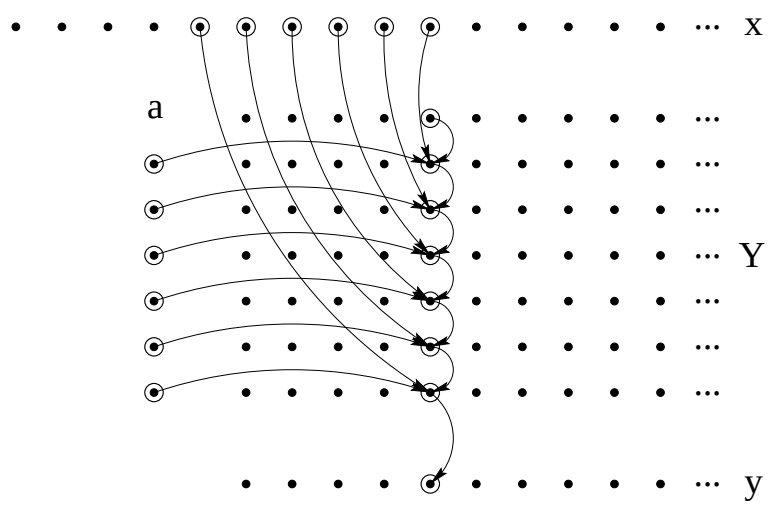

Figure 1: (partial) Dependence graph for the convolution filter

Now, if we consider N as a parameter, let us take any bijection ψ from $\{(i,N) \mid i \geq 0, N \geq 0\}$ to $\mathbb{N}$. For any $t \in \mathbb{N}$, we just take the same definition for $\varphi(t)$ as in the first case, with $(i,N) = \psi^{-1}(t)$.

The second example is one trivial case of a non finitely closed program.

```
system idc( a : { i | 0<=i } of integer )
       returns ( x : { i | 0<=i } of integer);
    var
       X : { i | 0<=i } of integer;
    let
       X[i] = a[i] + X[i+1];
       x[i] = X[i];
    tel
```

As we can see on Fig. 4, a parametrization of the dependence graph should give a subset containing one or more variable $x[i]$. As a consequence, this subset should also contain all $X[j]$ for $j \geq i$, and it cannot be finite.

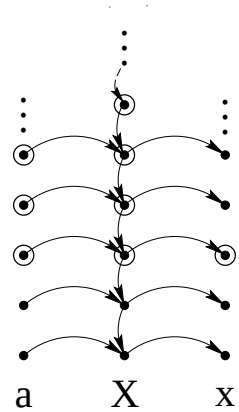

Figure 2: Dependence graph for the idc program

5 Application: proving the equivalence of ALPHA programs

In section 3.7, we used the notion of unambiguous invariant to prove the completeness of our proof method. This notion turns out to be valuable for proving equivalence between programs.

We first define a precise notion of equivalence between programs. This notion applies to programs having the same input and output, and is relative to some precondition and some set of output variables.

Definition 16 (Equivalence between programs) *Let S and S' be two programs having the same input and output sets In and Out. Let P be an assertion on input variables, and let V be a finite subset of Out. We say that S and S' are* equivalent from P with respect to V, *which is denoted $S \underset{P,V}{\equiv} S'$ iff for all $\sigma_0 \in Init$ such that $\sigma_0 \models P$, if $\sigma_0 \underset{V}{\Longrightarrow}^{S} \sigma$ and $\sigma_0 \underset{V}{\Longrightarrow}^{S'} \sigma'$ are two transition sequences, respectively of S and S', such that $V \subseteq supp(\sigma)$ and $V \subseteq supp(\sigma')$, then $\sigma \underset{v}{\sim} \sigma'$.*

Intuitively, the notion of unambiguous invariant means that such an invariant contains enough information to describe the results of the program. As expressed by the following fact, two programs respecting the same unambiguous invariant are equivalent.

Proposition 9 *Let S and S' be two programs having the same input and output sets In and Out, let P be an assertion on input variables, and let V be a finite subset of Out.*

If there exists I, unambiguous invariant for S and S' w.r.t. P and V, then $S \underset{P,V}{\equiv} S'$.

Proof ———— *Straightforward, from the definition of an unambiguous invariant.* ———— ❑

To prove that two programs are equivalent we just have to check that they respect the same unambiguous invariant. This can be done in two ways:

- Finding an invariant and check it for the two programs.
- Checking that one of the programs respects the canonical invariant of the other one.

Checking that a program respects some invariant can naturally be done "by hand": starting from some environment σ derived from an initial state and assuming that the considered invariant is true in σ, we have to consider all possible transitions for the program and prove that the invariant remains true in the resulting state. But let us remark that if I is an invariant for some program S, and if $I \Rightarrow I'$, then I' is also an invariant for S.

This adds thus a third possible way:

- Check that a unambiguous invariant of one program (its canonical invariant, for instance) logically implies an unambiguous invariant of the other one (again its canonical invariant, for instance).

Let us take an example to illustrate this point. We want to prove that the three following programs are equivalent for any subset of their output variables.

```
system S1 ( a : { i | i >= 0 } of integer; )
       returns ( x : { i | i >= 0 } of integer; )
    var
       X : { i | i >= 0 } of integer;
    let
       x[i] = X[i];
       X[i] = a[i];
    tel

system S2 ( a : { i | i >= 0 } of integer; )
       returns ( x : { i | i >= 0 } of integer; )
    var
       Y : { i | i >= 0 } of integer;
    let
       x[i] = Y[i];
       Y[i] = a[i];
    tel

system S3 ( a : { i | i >= 0 } of integer; )
       returns ( x : { i | i >= 0 } of integer; )
    let
       x[i] = a[i];
    tel
```

Let I_1 be the canonical invariant for S_1 with precondition *true*.

$$\begin{aligned} I_1 \quad \equiv \quad \forall i : \quad & x[i] = X[i] \wedge X[i] = a[i] \\ & \wedge\, \Delta(x[i]) \Rightarrow \Delta(X[i]) \\ & \wedge\, \Delta(X[i]) \Rightarrow \Delta(a[i]) \end{aligned}$$

It is straightforward to check that

$$\begin{aligned} I_1 \quad \Rightarrow \quad \forall i : \quad & x[i] = a[i] \\ & \wedge\, \Delta(x[i]) \Rightarrow \Delta(a[i]) \end{aligned}$$

the latter formula being the canonical invariant of S_3 with precondition *true*.
More generally, if some invariant contains

$$\begin{aligned} \forall i : \quad & X[\vec{i}] = E(Y[f(\vec{i})]) \\ & \wedge\, Y[\vec{i}] = F(Z[g(\vec{i})]) \\ & \wedge\, \Delta(X[\vec{i}] \Rightarrow \Delta(Y[f(\vec{i})]) \\ & \wedge\, \Delta(Y[\vec{i}] \Rightarrow \Delta(Z[g(\vec{i})]) \end{aligned}$$

these equations may be replaced by

$$\begin{aligned} \forall i : \quad & X[\vec{i}] = E(F(Z[g(f(\vec{i}))])) \\ & \wedge\, \Delta(X[\vec{i}] \Rightarrow \Delta(Z[g(f(\vec{i}))]) \end{aligned}$$

which is implied by the first formula.

Now, we can apply the same transformation to the canonical invariant I_2 of the second program. This yields the same formula. We thus have proved that

$$I_1 \Rightarrow I_3 \quad \text{and} \quad I_2 \Rightarrow I_3.$$

Now, since S_1 is equivalent to S_3 and S_2 is equivalent to S_3, S_1 and S_2 are equivalent.

The manipulation on invariants made above may be reflected by a manipulation on the program's text: it reduces to a textual substitution of definition expressions, suppressing intermediate expressions. This illustrates the duality between logic –canonical invariants– and syntax –program text.

6 Discussion

Our goal is to provide the ALPHA language with an external logical framework to prove functional properties of programs. The key idea is to associate to each program a logical formula, which captures its functional behavior without ambiguity. It is called its *canonical invariant*. This formula can be seen as a kind of "weakest invariant" for the program. Proving a property which does not depend on the operational scheduling of the program amounts thus to checking that it is logically implied by this invariant.

Because of the very simple structure of the ALPHA programs, we can give an explicit form for the weakest invariant. Surprisingly enough, it is essentially the same as the program text, up to the semantics of the "=" symbol. This is the reason why the refinement calculus on ALPHA program is so rich: most legal manipulations on logical predicates can be carried out directly at the level of programs.

We have moreover exhibited a number of sufficient conditions for a predicate to be an unambiguous invariant of an ALPHA program, if not its weakest canonical one. In some cases, for instance of the property under study refers to only a subset of the program variables, it may thus be possible to use a simpler invariant: simple properties can be proved by simpler proofs. Also, we have shown that usual methods for the equivalence proof of ALPHA programs are in fact special cases of such use of invariants. Again, the syntactic identity between a program and its canonical invariant is the key.

This preliminary work needs to be extended into several directions.

- Explore the equivalence proofs of ALPHA programs as found in the literature, and recast them into this logical framework.
- Understand more precisely how simple properties can be proved by simpler proofs. For instance, if a property is only concerned with a subset of the program variables, define a kind of "partially unambiguous invariant" which refers only to them.
- Understand how the logical framework sketched in this paper leads to revisit the refinement techniques for ALPHA programs. For instance, it would be interesting to design techniques to "project" an ALPHA program onto a subset of its variables, stripping out all irrelevant details.

Acknowledgments It is a pleasure to acknowledge many fruitful encouragements and discussions with Sanjay Rajopadhye on this work.

References

[1] C. Dezan and P. Quinton. Verification of regular architectures using Alpha: a case study. Internal publication 823, IRISA, Campus de Beaulieu, Rennes, France, 1994.

[2] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14(3):563–590, July 1967.

[3] C. Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures systoliques*. PhD thesis, Univ. Rennes I, France, December 1989.

[4] L. Nédelka. Étude expérimentale des systèmes de transitions des programmes Alpha. Master's thesis, IRISA, Campus de Beaulieu, Rennes, France, 1995.

[5] S. V. Rajopadhye. An improved systolic algorithm for the algebraic path problem. *INTEGRATION: The VLSI Journal*, 14(3):279–296, Feb 1993.

[6] D.K. Wilde. The Alpha language. Technical Report 999, IRISA, Campus de Beaulieu, Rennes, France, January 1994.

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