Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Cognitieve Kunstmatige Intelligentie

Programs, Interfaces and Components

Jan Bergstra *Marijke Loots*

Preprint nr. 006 July 1999

Artificial Intelligence Preprint Series

ISBN 90-393-2180-9
ISSN 1389-5184
Prof.dr. A. Visser, Editor

Onderwijsinstituut CKI
Utrecht University
Heidelberglaan 8
3584 CS Utrecht
The Netherlands

PROGRAMS, INTERFACES AND COMPONENTS

J.A. Bergstra & M.E. Loots, Eds.

March 28, 1999

Program Algebra & Projection Semantics

Interface Polarization & Coprogram Calculus

Program Aggregates & Coprogram Drivers

Program Algebra for Component Code

J.A. Bergstra* & M.E. Loots†

March 30, 1999

Abstract

The jump instruction is considered essential for an adequate theoretical understanding of imperative sequential programming. Using atomic actions and tests as a basis we outline an algebra of programs, denoted PGA, which captures the crux of sequential programming. PGA provides an ontology for programs rather than a semantics. Out of a multitude of conceivable semantic views on PGA we single out a semantical abstraction operator which assigns to each program a behavior. The meaning of the constants of PGA is explained in terms of behavioral abstraction semantics. Based on program algebra we define the general concept of a sequential programming language.

1 Introduction

The objective of this paper is to describe in a rather 'non-formal' and theoretical style the most simple and basic concept of a programming language that we have been able to discover. We have primarily focussed on sequential programming, because it still seems to be the basis of programming, both in practice and in teaching.

The target subject 'component code' indicates that the programs we will be able to formalise by means of our program algebra are rather low level in nature. Software components are closer to compiled executables than to human-readable high-level programs (we refer to [13] for the concept of a software component). The program algebra proposed in this paper will bear on disassembled components, if we may permit ourselves to use this trendy terminology.

The primary application of our work is to be found in teaching, where the basics of programming are often taught on the basis of full-sized programming languages, such as C, Pascal, Java or ML. Students of computer science will be the first to notice that a semantic description of programming languages is too

*University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl. Jan Bergstra acknowledges a significant critical remark from Anne Kaldewaij, as well as a number of suggestions for clarification of the text, many useful comments from Dimitri Hendriks, substantial suggestions by Alban Ponse and encouraging signs from Paul Klint.

†Tilburg University, Faculty of Philosophy, email: M.E.Loots@kub.nl.

complex to be taught or understood, in whatever curriculum one may imagine. Providing a clear semantic model of Java is still a matter of ongoing specialistic research, semantic models for ML and C being difficult as well. Even Pascal, which has been so popular in the world of teaching, is a non-trivial subject for semantic description.[1] As a consequence, semantic modeling has become a specialist activity, disconnected from the pragmatic world of programmers.[2] This is unfortunate because programmers may benefit from an understanding of the semantics of programming languages. Our approach is to introduce a very simple program notation. It is called PGA for ProGram Algebra[3]. In fact it is a format for languages; individual languages that comply with the format are to be found by instantiating the parameter set: the collection Σ of (boolean-returning) actions.[4]

1.1 Selfcontained exposition

It has been an important objective to have the main text of this paper self-contained in the sense that only a minimum of knowledge is presupposed. For instance, it is obvious that bisimulation has a role to play in the exposition of behavioral equivalence, but for reasons of simplicity we only pay attention to a version of trace theory. For the same reason, we have not proceeded along the lines of the metrical theory of behaviors developed by de Bakker and Zucker, nor have we used decorated traces from the CSP tradition. For the same reason we have decided not to elaborate on fully abstract models of program objects. Indeed, an object of writing this paper is to find out what may constitute a minimum of background knowledge on which a most simple discussion of a programming language and its meaning can be based.

1.2 Justification for PGA as a program notation

It will turn out that PGA-expressions can be used to express computer programs. Nevertheless, using PGA notation as a programming language will prove to be a very 'Spartan' exercise. The human programmer is likely to experience a significant incentive to use a richer or more high-level program notation. If PGA is not a notational format to be used by human beings, one may question its very classification as a programming language. In [7] it was put forward that programming is an activity performed by human beings, making the classification of PGA as a programming notation even harder. PGA is claimed to be a program notation on the following grounds:

[1] An obvious reference to what can be achieved in a classical semantical style is [8].

[2] We mention [9] as an interesting example of such work.

[3] For terminological reasons we will refer to the programming language implicit in PGA as PGLA. This is explained in detail in 6.2.

[4] PGA permits modifications that can handle void actions, i.e. actions that do not return any value with immediate impact on program control, or natural number-returning actions, or actions that return exceptions.

1. PGA programs are very close to stylized Assembly programs. Programming at that level has once been, and is still feasible as, a human activity.

2. When stored in a computer, PGA programs may well satisfy the criterion of implicit software from [7]. For that criterion to be satisfied, the steps of some running machine must be causally related to (or rather governed by) the program under consideration. Furthermore, if the program is replaced by another one in the same notation, the machine should start to act according to the new program. The latter requirement is called the flexibility criterion in [7].

3. PGA-notation is a potential carrier for manifest software, again using the terminology of [7]. We discuss this concept in more detail below.

For PGA we will provide one semantic model which maps programs onto behaviors. What a behavior stands for will only be sketched in vague outline. Behaviors have been studied in great detail in the literature in mathematical software engineering.[5] We will not discuss the mathematical notion of a behavior, in very much the same way as a book on applied calculus may fail to contain satisfactory definitions of the real numbers, and of continuity and discontinuity. The two cases are comparable in the sense that in both cases there is some mathematical substrate on which such a definition could be based.[6] PGA is so simple that it can be easily memorized together with its semantic equations.

Based on PGA more complex programming languages can be introduced. A reason for doing so is to introduce a firm semantic basis for such languages against minimal overhead costs. We will return to that matter in section 6.1.

1.3 Why Theoretical?

The phrase theoretical, as used in the abstract of this paper, requires careful motivation. In our view computer science sits between social and exact sciences. In the exact sciences the default meaning of 'theoretical X' is: 'mathematical X'. In the social sciences, however, 'theoretical X' stands for the philosophical methodological and conceptual basis of X. By positioning computer science in between, it follows that the meaning of theoretical computer science, and the inherited default meaning of 'theoretical X' for subtopics X of computer science (such as for instance: ProGram Algebra) may still be a matter of free choice. In contrast with the currently used default we propose that for computer science and for its subtopics the default meaning of 'theoretical X' is inherited from the gamma and alpha sciences. As a consequence, theoretical PGA is primarily

[5] Often behaviors are discussed in terms of processes. Stated differently: processes are behaviors, and exhaust the behaviors by and large.

[6] In fact the mathematical definition of a behavior is not at all a necessary richness of the mind when the notion is used for standard semantic questions. In just the same way as one can explain elementary calculus to those who fail to understand Cauchy sequences one can explain a behavioral semantic model for an elementary programming languages to those who fail to understand transition systems modulo bisimulation.

used to denote a philosophical and 'non-formal' discussion of the principles of PGA. Mathematical content matter is to be minimized as much as possible to leave ample scope for 'non-formal' conceptual development.

In [7] a general division of software engineering (SE) is proposed in TSE (theoretical SE), MSE (Mathematical SE), ESE (Experimental SE) and PSE (Practical SE, itself the union of DPSE or descriptive PSE and FPSE or field PSE[7]). Theoretical PGA or TPGA can be classified under TSE. Of course, a more mathematical discussion of PGA and its extensions is possible. Such developments will be referred to as developments in Matematical PGA or MPGA[8] and will be classified under MSE.

1.4 Manifest Program Notations

In [7] a division of software into four categories is proposed: implicit software, explicit software, hypothetical software and manifest software.

Implicit software is stored in a memory compartment of a machine. For a data compartment in a machine to contain implicit software it is essential that some behavior of the machine can be understood to be causally dependent on the memory segment in question. Furthermore, the dependency of the machine behavior should be considered flexible. This means that by appropriately changing the contents of the memory compartment containing the program, a very wide range of program behaviors can be obtained. Loaded executables are the prime examples of implicit software. It is theoretically possible for implicit software to originate from a process without human involvement.[9]

Explicit software is text for which a standard translation into implicit software is widely available. Well-known programming languages such as COBOL and C allow one to produce such explicit software.

Hypothetical software is a textual product, which can be transformed into a loaded executable under limited conditions. Programs expressed in languages for which compilers have not yet been written, or are speculative altogether, are classified as hypothetical.

Manifest programs are like mathematical notations: independently of any technological support they will always be recognised as instructions for a machine.

Few program notations can qualify as generating manifest software[10]. It is our objective to introduce with PGA a notational format that may become so widely accepted that its classification as manifest program notation (i.e. a notation in which manifest software can be encoded) is reasonable. For that classification to be reasonable, the distance from the technology that is used

[7] In Dutch this category is called UPSE or 'Uitvoerende PSE'.

[8] This may sound odd as program algebra by itself sounds 'mathematical'. However, if one views logic as a branch of philosophy 'mathematical logic' has a comparable status and a similar remark may be made in connection with the phrase 'mathematical statistics'.

[9] All programmers produce software, but not all software is produced by human programmers.

[10] A convincing example of manifest software is the pure lambda calculus ([2]).

at any stage must be so significant that the notation does not run the risk of becoming outdated. This very distance is a property of many mathematical notations. We claim that PGA satisfies this criterion. Other notations satisfying the criterion are: flow charts for deterministic sequential programming, complete term rewriting systems for functional programming, process algebra notation for concurrent programming.

We claim that a manifest program notation largely independent of technology should underly the teaching of computer programming. The claim that such notations are in short supply may be hard to substantiate. To the best of our knowledge what we propose is new.[11] The viewpoint that the simplest program notations are to be found as instantiations of a single extremely simple schema is both attractive and risky. In particular we have to accept non-structured control mechanisms (in this case the jump instructions) as a basis, and should not be afraid of using a non-compositional semantics.

In the case of PGA we will outline behavioral equivalence as a semantic view. Behavioral equivalence fails to be compositional because two programs that are behaviorally equivalent may show a difference when embedded in the same context.[12] We claim that the use of non-compositional semantics in connection with an abstract assembly language provides a very useful point of departure for the understanding of both the meaning and usage of programming languages. The objection that 'all programming must be done at a higher level of abstraction' is less and less clearly a refutation of our claim.

2 Program Expression Syntax

The syntax of program expressions in PGA is generated from six constants (or kinds of constants) and two composition mechanisms. The constants are made from a parameter set Σ of so-called atomic instructions or atomic actions.[13] These atomic actions may be viewed as requests to an environment to provide some service. It is assumed that upon every termination of the delivery of that service, some boolean value is returned that may be used for subsequent program control. The constants can be viewed as basic instructions. The composition mechanisms of program algebra are the structuring 'features' of the programming language. The compositions are:

- text-sequential composition of X and Y, written $X;Y$, and
- repetition of X, written X^ω.

[11] Of course the phrase program algebra, or algebra of programs is by no means new. It was clearly advocated by Backus in his Turing award lecture. Each instance of the phrase algebra of programs known to us is found in the context of structured and functional programming constructs, however, whereas we will primarily focus on non-structured control mechanisms. As a consequence we have to face non-compositional semantic modeling.

[12] For an example in detail see 4.5.3.

[13] Atomic actions are considered indivisible from the perspective (abstraction level) of the program. At a lower level of abstraction the atomic actions may comprise entire program executions for other programs.

The constants, or primitive instruction types, are listed below, the negative test intructions having been included for the reason of symmetry. Indeed we consider the pair true and false symmetrical in PGA:

void atomic instruction. All elements $a \in \Sigma$ are basic instructions; when executed these actions may modify (have a side-effect on) a state, a boolean value being generated in addition. The attribute void expresses that these instructions do not make use of the returned boolean value. After having performed an atomic instruction a program has to enact its subsequent instruction. If that instruction fails to exist, an error occurs (or: meaningless behavior is produced).

termination instruction. The instruction ! indicates termination of the program. It will not lead to any further effects on the state, and it will not return any value.

positive test instruction. For all actions $a \in \Sigma$ there is the positive test instruction $+a$. If $+a$ is performed by a program, the state is effected according to a, after which the sequence of remaining actions is performed in case true was returned. If there are no remaining actions an error occurs. If false was returned after a was performed, the next action is skipped and execution proceeds with the instruction following the instruction that was skipped. In that case (i.e. when false was returned) at least two instructions must be present following $+a$; otherwise an error will occur.

negative test instruction. For all actions $a \in \Sigma$ there is the negative test instruction $-a$. If $-a$ is performed by a program, the state is effected according to a, after which the remaining sequence of actions is performed in case false was returned. If there are no remaining actions an error occurs. If true was returned after a was performed, the next action is skipped and execution proceeds with the instruction following the instruction that was skipped. In that case (i.e. when true was returned) at least two instructions should be present following $-a$; otherwise an error will occur.

forward jump instructions. For any natural number k there is an action $\#k$ which denotes a jump of length k. We call k the counter of the jump instruction. If $k = 0$, this jump is to the instruction itself (zero steps forward). In this case a loop or divergence will result. If the counter of the jump equals 1, the instruction is a skip (i.e. it skips itself). If the counter (k) exceeds 1, the effect of the execution of an instruction $\#k$ is to skip itself and the next $k - 1$ instructions. If there are not that many instructions left in the remaining part of the program, an error will result.[14]

[14]For natural number k, $\#k$ is a static jump instruction. A dynamic jump can be imagined just as well. Let c be an action that returns a natural number, then $\#'c$ is a dynamic jump. Its execution will proceed as follows: c is performed, returning k. Then $\#k$ is performed. We will not discuss the technical aspects of dynamic jumps in this paper.

abort instruction. The jump without a counter, #, denotes the instruction which aborts a computation due to an error. In contrast with !, # constitutes an unsuccessful termination.

Examples of programs (or rather program expressions) are: $a; b; c$, $b; +a; \#5; \#2; c; c$, $(a; -b)^\omega; \#2; !$, and $(\#1)^\omega$. Capitals $X, Y, Z, U, ..$ (often used with subscripts and/or superscipts) will be used as variables for programs. The same capitals will also be used to range over program expressions.[15] Atomic instructions are denoted with $a, b, c, ..$ often being decorated with subscripts or superscripts as well.

2.1 The program notation PGLA

With a program or program expression we will denote any closed term[16] over the above syntax. The collection of these is denoted with PGA(Σ). In view of the intention to provide a systematic development of program notations we will view PGA as a programming language in its own right. That language will henceforth be denoted with PGLA. The programs that can be written using atomic actions in Σ are collected in PGLA_Σ.

DESIGN RATIONALE for PGLA: PGLA has been designed to achieve the following design objectives:

1. PGLA should be an expressive program notation that can hardly be simplified from a conceptual point of view. It should be easy to teach. PGLA should have no bias towards any notation for data manipulation. PGLA syntax should be of minimal complexity.
2. Every state of a PGLA program corresponds to a PGLA program as well. (The corresponding property for processes is characteristic of all process algebras.)
3. PGLA features should be semantically orthogonal (as much as possible); as a consequence the definition of behavioral semantics for PGLA should be straightforward.
4. PGLA complies with the principal identification of a program as a sequence of instructions.

3 Program Object Equations

The program expressions can be interpreted as simply identifying programs that give rise to identical sequences of instructions. Admittedly this is a rather superficial ontology. In fact far more sophisticated identifications can be made. In

[15] A systematic distinction between variables and meta-variables has not been made in order to simplify the presentation. This additional precision can easily be introduced by taking $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{U}, ...$ as meta-variables ranging over program expressions.

[16] A closed term is an expression without variables.

the next section we will outline a notion of behavioral equivalence for programs. It is possible to identify programs as long as that identification does not imply that two programs with different behavior will have to be considered identical.[17] It is easy to define for $n > 0$, the n-th instruction of a program (if it does not exist, that is, if the program is too short, we take # as the n-the instruction). Equally easily one can define $icnt(X)$ the number (count) of instructions of a program X. In the case of a repeating program that number is infinite.

An important principle for program equivalence is AIP[18]: if X and Y have equal length and equal n-th instructions for all natural $n > 0$, then the two programs must be equal. As a notation for the n-th instruction[19] of program X we propose: $i_n(X)$.[20] We present 4 equations and a conditional equation that identify representations of the same sequences of instructions:

$$(X;Y);Z = X;(Y;Z)$$

$$X^\omega = X;X^\omega$$

$$X^\omega;Y = X^\omega$$

$$(X;Y)^\omega = X;(Y;X)^\omega$$

$$X;Y = Y;X \rightarrow X^\omega = Y^\omega$$

The associativity of concatenation implies (as usual) that far fewer brackets have to be used. We will use[21] associativity whenever confusion cannot emerge. The expansion of repetition, as expressed in the second equation, is needed to explain which sequence of actions is in fact generated by a repeated program. It is then obviously impossible to perform infinitely many actions in finite time[22]. It follows that right cancellation is obtained if a left-hand argument of a concatenation contains an infinite repetition. This is expressed by the third equation.

[17] This is a general construction: given an equivalence on an algebra one may define the largest congruence on the algebra which is compatible with that equivalence. Albert Visser has pointed out to us a nice representation of the fully abstract model in terms of a decorated state transition system.

[18] This name is taken from the terminology of Process Algebra [1]. AIP stands for Approximation Induction Principle and has a similar form. Indeed AIP will allow one to use induction on approximations to prove the identity of programs. For instance: $a;(b;a)^\omega = (a;b)^\omega$ can be shown using AIP and induction on the variable k mentioned in AIP. Another important consequence of AIP is: $X^\omega = (X;X)^\omega = (X;X;X)^\omega = ...$ We notice, however, that these identities can also be derived from our system of 5 (conditional) equations.

[19] Here n must be a positive natural number.

[20] Defining equations for this operator: $i_1(u) = u$, $i_1(u;X) = u$, $i_{n+1}(u) = \#$, $i_{n+1}(u;X) = i_n(X)$.

[21] The use of associativity consists of the deletion of brackets that are necessary in case an operator fails to be associative. Explicit presence of redundant brackets is always legal of course.

[22] So-called Zeno executions are excluded. This point requires some care. The exclusion of Zeno executions regards computations in which an action is preceded by infinitely many actions. Such computations are excluded in many theories of computation on philosophical grounds. In theories not based on a total ordering of time the concept of a Zeno execution should be phrased in terms of causal precedence rather than temporal precedence. It is a consequence of the very restricted syntax of PGLA that a computation of a program containing infinitely many actions must inevitably involve infinitely many steps. PGLA contains no jump instruction enabling it to 'jump over' an infinite sequence of instructions in one step.

3.1 Structural congruence

Two closed program expressions X and Y are called program object equivalent if their equivalence follows from the program object equations above. As an alternative terminology 'structural congruence' can be used for program object equivalence. This terminology has been introduced by Milner in connection with the famous π-calculus. Structural congruence identifies programs that will certainly have the same meaning. Structural congruence is used to simplify subsequent discussions regarding program semantics. We notice that program object equivalence is a decidable matter for closed program expressions.

3.2 Canonical forms for PGA: the first canonical form

Program transformations will be very important below. It turns out that many program transformations profit from a preprocessing phase in which a program to be transformed is first rewritten into a simpler, but structurally congruent, form.

Let X be a closed PGA expression. Then X can be rewritten into one of the following forms:

- Y, not containing repetition,
- Z^ω, with Z not containing a repetition, or
- $Y;Z^\omega$, with Y and Z not containing a repetition.

In any of the above forms, we will call closed terms canonical terms. The subterm Z in the second and third case is called the repeating part of the (canonical) expression. The proof of the existence of the first canonical form uses induction on the structure. The key case is this: $(X^\omega)^\omega = X^\omega; (X^\omega)^\omega = X^\omega$. Examples of identities between terms and terms in the first canonical form:

- $a^\omega; b; !; c = a^\omega$,
- $+b; (!; c; -b; (c; \#25; a; !)^\omega)^\omega = +b; !; c; -b; (c; \#25; a; !)^\omega$.

Confusion may arise around the notion of a repeating part and that of periodicity. The repeating part is a portion of a syntactic expression. A program object, however, is periodical if it is the interpretation of a canonical term with infinite length. Its period is the shortest length of a repeating part of a canonical form representing the object.

3.3 Canonical forms for PGA: the second canonical form

There is an important subclass of the canonical terms, the so-called second canonical forms, for which the following additional requirement holds: no counter used for a jump in the form exceeds or equals the instruction length of the repeating part.[23] We assume that we consider a canonical form of type (iii) in

[23] All programs without repetition operator are also in second canonical form.

the listing above. The second canonical form W is obtained as follows: let k be the instruction length of the repeating part Z in the canonical form. Let l be the maximal counter of a jump used in the canonical form. Choose m minimal such that $l < m \cdot k$. Then let $U = X^m$ and the required second canonical form is
$W = Y; U^\omega$. The program X^m must still be defined (for $m > 0$). It is X if $m = 1$, and $X; X^{m-1}$ if $m > 1$. Examples of transformations into the second canonical form:

- $a; \#9; (+b; !; c)^\omega = a; \#9; (+b; !; c; +b; !; c; +b; !; c; +b; !; c)^\omega$,
- $-a; b; (+c; \#8; !)^\omega = -a; b; (+c; \#8; !; +c; \#8; !; +c; \#8; !)^\omega$.

4 A behavioral abstraction operator

In this section we will heavily rely on the concept of a behavior. We will refrain from providing a detailed mathematical theory of behaviors and use the concept as an intuitive notion instead. Thus, we need an informal description of the concept of a behavior. The basic intuition as regards program behavior concerns the external effect of a program in contrast to its internal workings. The behavior of a program abstracts from all internal mechanics, focussing entirely on aspects of immediate relevance for the external behavior of a program. Each time an atomic instruction is performed some external effect may take place. The temporal ordering of instruction execution must be displayed in the behavior, while forgetting the program internals causing that ordering.

We use behaviors to describe the (potential) external effect, or, in our case, the (potential) effect on a state space, of a program. The internal workings, which have to do with the bookkeeping of returned boolean values and the effect of jump instructions, are to be abstracted from when determining the behavior of a program. We consider behaviors to be generated by means of composition mechanisms from primitive or irreducible behaviors.

4.1 Behavior composition mechanisms

For a program object X the object $|X|$ denotes its behavioral abstraction. We will write P and Q for behaviors. Two compositions for behaviors are fundamental for our purposes: sequential composition and conditional composition.

$P \cdot Q$ denotes the sequential composition of P and Q. A behavior terminates after its last action is performed. In a sequential composition the first behavior performs all of its actions, after which the second behavior will produce a complete sequence of its actions. A behavior may continue with different actions from the same state. A most typical example of such a situation is found when a behavior contains an action $a(-)$, which can have various parameters. Then a parameter may indicate a value which the behavior will obtain from its environment. It is then reasonable to assume that from a certain state in the behavior, actions $a(d)$ occur, for different parameter values d.

The second composition principle that is essential for behaviors is the conditional composition. Using the notation of [11] we denote with $P \lhd \phi \rhd Q$ the behavior which proceeds as P if the condition ϕ is valid and which behaves like Q if the same condition fails to hold. Notice that the condition ϕ may depend on a context, state or environment (depending on one's favorite terminology).

4.2 Irreducible behaviors

Irreducible or atomic behaviors are primitive notions in our semantic view. Although in practice these atomic behaviors may themselves be decomposable into smaller units of operation, in our case we treat them as irreducible ones. There are four such irreducible behaviors:

atomic action. For $a \in \Sigma$ the behavior $||a||$ corresponds to performing a. The execution of a yields a boolean value. We notice that $|a|$ and $||a||$ are to be distinguished: $|a|$ is a form to be evaluated whereas $||a||$ is a result description.

In an interpreted case the actions will have side effects on a state. A side effect is just a change in state. We will provide no further details on these side effects other than stating that performing an action must always be possible[24] and that it is not essential to require the effect of an action to be deterministic.

termination. With $||!||$ terminating behavior is denoted; it does no more than terminate, after which control is handed over to a subsequent behavior (if any). Termination actions will not have any side effect on a state.

divergent behavior. With D a divergent behavior, also called a loop, is indicated. It is a behavior that represents an internal cycle of activity without any external effect whatsoever.[25] Like termination behavior, a loop behavior does not affect a state in which it occurs.

meaningless behavior. With M we denote the behavior of a system which runs into an error.[26]

Using these irreducible behaviors more complex behaviors can be composed by means of sequential and conditional composition.[27]

[24]The assumption that all actions are non-blocking serves as a simplification. Program algebra can be generalized to deal with blocking actions as well.

[25]A loop occurs in case an infinite number of consecutive jumps is performed.

[26]The notations D and M have been taken from [3] where they denote additional truth values in a four-valued logic with exactly the same intuitive explanation.

[27]Process Algebra is a mathematical theory of behaviors. ACP style process algebra was introduced in [4]. Connections of ACP with programming methodology were explored in [6]. A treatment of process algebra beginning with the composition operators listed here (sequential composition and conditional composition), and both special constants D (written δ) and M (written μ) can be found in [5]. A different approach to the modeling of sequential programming via ACP style processes can be found in [10].

4.3 Conditions

For a semantic interpretation of the programs of PGLA_Σ just one kind of conditions is needed. Let a be an action in Σ. We write $Y(a)$ for the condition which holds true exactly if action a, when performed, will yield the boolean value true. Otherwise $Y(a)$ will not hold and a will produce value false.

For this kind of condition to be meaningful, it is essential that in a context the boolean value yielded by the action a is unambiguously determined. This is a prerequisite for the use of the functional notation $Y(a)$. It should be stressed that evaluation of the condition $Y(a)$ does not by itself entail that a is performed. Indeed $Y(a)$ is a theoretical term, a hypothetical condition, which may be very hard to implement in a practical instance. Nevertheless, if a can be assumed to yield a unique boolean value there can be no objection against the introduction of a name for that value. The actual truth value that $Y(a)$ denotes will in general depend on the state of the context in which a program is being executed.

It is important to notice that an action which returns a boolean value can have a non-deterministic side effect. This is entirely consistent with its yield being determined in advance. The effect on the state is more or less independent of the yield. All conditions ϕ that will be used in the behavioral semantics of PGA are of the form $Y(a)$ for some action $a \in \Sigma$.

4.4 Program Behavior Equations

Semantic equations will describe the behavior of complex programs in terms of the behavior of their constituent parts. Behavioral semantic equivalence will not be compositional. We will comment on compositionality below. Technically speaking the semantic equations amount to nothing more than a definition of the behavioral abstraction operator $|-|$ on PGA-objects.

In combination with an intuitive understanding of a behavior, the composition operators described for behaviors and of the irreducible behaviors, the semantic equations provide memorisable detail for the intuitions given for the building blocks of PGA. By memorising (and understanding) the semantic equations one may reasonably claim to know the semantics of PGA. Of course, it should be kept in mind that the semantic equations below provide only one out of very many semantic options. In order to fully understand these equations, one should notice that there are no overlaps leading to ambiguities, and that each closed program expression will match one of these expressions. One may view the equations as defining equations for the behavioral abstraction function. This definition uses induction on the number of instructions in a program and, on top of that, an induction on the counter of a jump instruction. In the latter induction three cases are distinguished: zero, one and above one.

In the equations below a ranges over an action in Σ, u ranges over all instructions and X ranges over arbitrary program objects.

4.4.1 Semantic equations for atomic actions and tests

The behavior $|X|$ of program object X is determined using a complex tail recursion. In all cases the behavior starts with performing the behavior of the first instruction, if possible. If the program ends without being able to perform an explicit termination instruction, an error occurs.

$$|a| = ||a|| \cdot M$$

$$|a; X| = ||a|| \cdot |X|$$

$$|!| = ||!||$$

$$|!; X| = ||!||$$

$$|+a| = |a|$$

$$|-a| = |a|$$

$$|+a; u| = |a; u| \lhd Y(a) \rhd |a|$$

$$|-a; u| = |a| \lhd Y(a) \rhd |a; u|$$

$$|+a; u; X| = |a; u; X| \lhd Y(a) \rhd |a; X|$$

$$|-a; u; X| = |a; X| \lhd Y(a) \rhd |a; u; X|$$

4.4.2 Semantic equations for the jump instructions

The case of the jump instructions requires a case distinction on the counter of the jump. In case that counter is zero, a divergence will occur. In case that counter is one, at least one further instruction should be performed, otherwise an error occurs. In case the counter exceeds two, the program should contain at least two subsequent instructions; otherwise the program will produce an error. In these equations k ranges over the natural numbers.

$$|\#0| = D$$

$$|\#0; X| = D$$

$$|\#1| = M$$

$$|\#1; X| = |X|$$

$$|\#k + 2| = M$$

$$|\#k + 2; u| = M$$

$$|\#k + 2; u; X| = |\#k + 1; X|$$

4.4.3 Equations for unsuccessful termination

There are only two cases:

$$|\#| = M$$

$$|\#; X| = M$$

4.4.4 Divergence for non-trivial loops

The above equations should be used to obtain successive steps of the behavior of a program object X. The equations may be applied infinitely often without ever generating a piece of atomic behavior. In that case the program has a non-trivial loop[28]. In this case it will be identified with D. By doing so we obtain for instance: $|(\#1)^\omega| = D$ and $|b;(\#2;a)^\omega| = ||b|| \cdot D$.

If this identification of behaviors exhibiting a non-trivial loop with D is omitted as a transformation rule, the set of all equations mentioned so far constitutes an orthogonal term-rewrite system.[29]

4.5 Program behaviors

Program behaviors can be finite and infinite. A behavior is called finite if there is a finite upperbound to the number of consecutive atomic action behaviors it can perform[30]. Of course the behavior of all programs not involving repetition is finite. Programs with repetition can have infinite behaviors. As an example consider $X = a^\omega$. We find $|X| = ||a|| \cdot ||a|| \cdot ||a|| \cdot ....$

Infinite behaviors can be approximated by finite behaviors. The n-th approximation of a given (finite or infinite) behavior is found by substituting D for each subbehavior that starts after n atomic action behaviors (or termination behaviors) have occurred. So the second approximation of $||a|| \cdot ||b|| \cdot ||a|| \cdot ||!||$ equals: $||a|| \cdot ||b|| \cdot D$. Of course the n-th approximation of X is always a finite behavior.

Equality of infinite behaviors can easily be retrieved from equality of finite behaviors. Two (finite or infinite) behaviors are equal exactly if for each natural number n, the n-th approximations of the two behaviors are equal. It follows that once it has been defined (or understood) when two finite behaviors are equal, that definition automatically extends to the case of infinite behaviors.

Finite approximations of programs are considered equal if and only if they have exactly the same form.

4.5.1 Examples

For a better understanding of the implications of the above semantic equations we provide a number of examples. It should be noticed that often the equations will not provide a recursion-free definition. Indeed, the semantic equations may give rise to infinite systems of equations. Such infinite systems have unique solutions just like finite systems.

Here are some consequences of the semantic equations above.
$|a;!| = ||a|| \cdot ||!||$, $|\#1;!| = ||!||$, $|\#2;!;!| = ||!||$, $|a^\omega| = ||a|| \cdot |a^\omega|$, and
$|+a;-b;c;!| = (||a|| \cdot (||b|| \cdot ||!|| \lhd Y(b) \rhd ||b|| \cdot ||c|| \cdot ||!||)) \lhd Y(a) \rhd ||a|| \cdot ||c|| \cdot ||!||$.

[28] The program #0 will lead to a trivial loop and its meaning can (but need not) be equated to D.

[29] See Klop[12] for a complete exposition of orthogonality and term rewriting.

[30] As a consequence D is considered a finite behavior.

4.5.2 Behavioral Equivalence

Two programs X and Y are behaviorally equivalent (denoted by or $X \equiv_{be} Y$) if $|X| = |Y|$. It can be shown that it is decidable whether or not $X \equiv_{be} Y$ for closed program algebra expressions X and Y. As an example we mention: $a; b; c \equiv_{be} a; \#2; \#1; b; c$.

4.5.3 Non-compositionality of behavioral equivalence

Non-compositionality of behavioral equivalence is understood via some simple examples as well. As an example consider $X = \#2; a; b; !$ and $Y = \#2; c; b; !$. It will be clear that $|X| = |Y| = ||b||\cdot||!||$. On the other hand $|\#2; X| = ||a||\cdot||b||\cdot||!||$ which differs from $|\#2; Y| = ||c|| \cdot ||b|| \cdot ||!||$.

4.5.4 Avoiding errors

The semantic equations certainly express that only explicit termination can terminate the execution of a program without running into an error. If one intends to guarantee that the behavior of a program X does not evolve into M (after a number of actions), it suffices to require first of all that X is denoted by an expression that does not involve $\#$ and secondly to postfix the program with an unbounded supply of termination instructions: indeed, for any such program expression X the object $X; !^{\omega}$ is M-free. By using programs of that form only, which of course will require some type checking, error behavior can be avoided.

5 Program behavior and input-output relations

For given Σ we define the notion of a deterministic state space as follows. A deterministic state space consists of a set V together with functions $\mathit{effect}_a(-) : V \to V$ for each action $a \in \Sigma$[31] , and a mapping $y_a(-) : V \to \{T, F\}$ for each $a \in \Sigma$. A behavior P determines a function of type $V \to V \cup \{D, M\}$, as follows. We will write $P \bullet v$ for the value of this function.[32] It represents what P computes on input v in V. The definition of '$\bullet$' is with induction on the top-level structure of P.

1. $M \bullet v = M$,
2. $D \bullet v = D$,
3. $||!|| \bullet v = v$,
4. $||a|| \bullet v = \mathit{effect}_a(v)$,
5. $(||a|| \cdot P) \bullet v = P \bullet \mathit{effect}_a(v)$
6. $(P \lhd Y(a) \rhd Q) \bullet v = (P \bullet v) \lhd y_a(v) \rhd (Q \bullet v)$.

[31] in some cases the notation $v_{\partial a}$ is used for $\mathit{effect}_a(v)$.
[32] A more specific notation for this operator $(P \bullet v)$ is $P \bullet^{io} v$.

Whenever these identities do not determine a value for $|X|\bullet v$ we take $|X|\bullet v = D$ in order to express that the computation diverges. For a given deterministic frame F one obtains an equivalence relation $\equiv_{ioF}$ on program objects which declares X and Y equivalent whenever the corresponding mappings $|X| \bullet (-)$ and $|Y| \bullet (-)$ coincide on all v in V.

6 Program Algebra Projections

A program algebra projection is a mapping from a set L into PGA(Σ) for an appropriate action set Σ. Such a mapping can be defined using many techniques. Evidently a program algebra projection φ turns objects in L into programs. It follows that using φ an algorithmic or operational meaning can be assigned to objects in L. One may write $|X|_L = |\varphi(X)|$ for X in L.

It may well be the case that a far more elegant way of assigning $|X|$ to X can be found than the detour via $|\varphi(X)|$. However, the detour is acceptable, and it does not commit one to any claim concerning the existence of high-level or denotational methods to obtain a behavioral semantics for objects in L.

6.1 Programming Languages

A programming language can be defined as a pair (L,φ) with L some collection of textual objects and φ a program algebra projection. In fact, it may be necessary (and is considered acceptable) to allow the use of auxiliary actions in the PGA into which the projection translates the objects in L. The most obvious example is the use of stack manipulation actions needed if L contains instructions manipulating nested expressions for mathematical values.

It is reasonable if not mandatory to require that the program algebra projection is a computable mapping. This definition is to be read as a criterion rather than as a dogmatically limiting constraint. If, for instance, a program algebra projection can only be made when the program algebra has been extended with dynamic jumps, that will not refute our approach. The present definition of a programming language is simplistic in nature but, at least in principle, it covers such illustrious examples as COBOL, Java, C, and C++, and less mature examples such as ASF+SDF, PROLOG and CLEAN.

The projection can be seen as a theoretical compiler, optimized for human understanding rather than for one of the more conventional objectives, such as: code-compactness (small expression that represents the output of the projection), target code space and/or time efficiency, and compiler execution space and/or time efficiency.

6.2 PGA as a program notation: PGLA

PGLA is just PGA viewed as a programming language. So it is the pair (clPGAexp, evalPGA) where clPGAexp represents all closed PGA expressions,

and evalPGA represents the interpretation or evaluation of closed PGA expressions into PGA-objects. PGLA being available, it is obvious that a program algebra projection for a set F can be represented by a mapping `f2pgla`, which maps F expressions into PGLA expressions. We will write $|X|_{pgla} =$ |evalPGA(X)|. Below program algebra projections will be provided via translations into PGLA. Such translations may take several steps through intermediate languages.

6.3 Avoiding aborts in mfPGLA

M-free PGLA, or mfPGLA, is the programming language determined by the pair (clPGA'exp, `mfpgla2pgla`), where clPGA'exp represents all closed PGA expressions not containing the abort instruction # and where the projection operator reads as follows: $\texttt{mfpgla2pgla}(X) = X;!^\omega$. The programs of mf PGLA cannot run into an error state (caused by a program jumping into non-existent lines of the program). For PGA programs that do not run into errors, the M-free PGA semantics has exactly the same behavior. It would have been quite possible to define PGA as if its semantics were that of mfPGLA. The reason not to do so is that it makes the language less orthogonal as termination can then result both explicitly as the result of a termination instruction and more implicitly by reaching the end of the program text.

6.4 Forward and backward jumps in PGLB

PGLB is an important variation on PGLA. It has an additional backward jump instruction written $\backslash\#k$. In the presence of backward jumps, repetition becomes a redundant feature, so the only composition of PGLB is concatenation (;). Each program is a sequence $u_1; ...; u_k$ of instructions.

The program algebra projection `pglb2pgla` is defined using auxiliary operations ψ_j. Given PGLB program $u_1; ...; u_k$ we write:

$$\texttt{pglb2pgla}(u_1; ...; u_k) = (\psi_1(u_1); ...; \psi_k(u_k); \#; \#)^\omega.$$

The operations ψ_j then read as follows:

$$\psi_j(\#l) = \#l \ \texttt{if}\ j + l \leq k$$

$$\psi_j(\#l) = \# \ \texttt{if}\ j + l > k$$

$$\psi_j(\backslash\#l) = \#k + 2 - l \ \texttt{if}\ l < j$$

$$\psi_j(\backslash\#l) = \# \ \texttt{if}\ l \geq j$$

$$\psi_j(u) = u \ \texttt{otherwise}.$$

Examples:

$$\texttt{pglb2pgla}(+a) = (+a; \#; \#)^\omega,$$

$$\texttt{pglb2pgla}(c; +a; !; \backslash\#2; \#5; -b; \#1; !) = (c; +a; !; \#8; \#; -b; \#3; !; \#; \#)^\omega.$$

6.5 Conventional termination in PGLC

PGLC is a minor variation on PGLB. PGLC has no explicit termination instruction. Termination takes place when the last action in the list has been executed (and was no backward jump) or when a forward or backward jump is made to an instruction outside the list. The program algebra projection `pglc2pglb` is given by:

$$\texttt{pglc2pglb}(u_1;...;u_k) = \psi_1(u_1);...;\psi_k(u_k);!;!$$

The auxiliary operators ψ_j are given by:

$$\psi_j(\#l) = ! \ \texttt{if} \ \ j+l > k$$

$$\psi_j(\backslash\#l) = ! \ \texttt{if} \ \ l \geq j$$

$$\psi_j(u) = u \ \ \texttt{otherwise.}$$

Examples: $\texttt{pglc2pglb}(+b) = +b;!;!$ and $\texttt{pglc2pglb}(+c;\#10;\backslash\#1;-c;\#2;+b) = +c;!;\backslash\#1;-c;!;+b;!;!$

6.6 Absolute jumps in PGLD

PGLD is yet another variation on PGLA. As in PGLC, repetition and explicit termination have been omitted. The termination conventions are the same as in PGLC. Instead of forward and backward jumps PGLD allows absolute jumps; $\#\#k$ for a natural number k, is an instruction which makes control move to (the beginning of) the k-th instruction of the program. If $k = 0$ termination will occur. We find a program algebra projection for PGLD by first projecting it into PGLC, and subsequently using the projection for PGLC.

The program algebra projection `pgld2pglc` is obtained as follows. Given PGLD program $u_1;...;u_k$ we define:

$$\texttt{pgld2pglc}(u_1;...;u_k) = \psi_1(u_1);...;\psi_k(u_k).$$

The operations ψ_j then read as follows:

$$\psi_j(\#\#l) = \#l - j \ \ \texttt{if} \ l \geq j$$

$$\psi_j(\#\#l) = \backslash\#j - l \ \ \texttt{if} \ l < j$$

$$\psi_j(u) = u \ \ \texttt{otherwise.}$$

Example: $\texttt{pgld2pglc}(a;+b;\#\#1;\#\#8;c;\#\#5;f) = a;+b;\backslash\#2;\#4;c;\backslash\#1;f.$

7 Conclusions

We have proposed PGA, an algebra in which an extremely simple programming language is captured together with an axiomatic semantic model in terms of behaviors. PGA is parameterised by a set Σ of atomic actions. Both PGA and the semantic equations for the features of PGA are very simple indeed and can easily be memorised. Particular programming languages can be developed either by instantiating the parameter set of PGA or by translating new syntax into PGA (with instantiated parameter).

The virtue of PGA is that it can be used to anwer the question 'what is a programming language' by providing a simple and general construction. It can be used to program Turing machines as well as to model simple assembly languages. The authors consider PGA to be a meaningful point of departure for the teaching of programming and software engineering.

References

[1] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In *Handbook of Logic in Computer Science*, volume IV, pages 149–286. Oxford University Press, 1995.

[2] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, Amsterdam, 1984.

[3] J.A. Bergstra, I. Bethke, and P. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied Non-Classical logics*, 5:199–217, 1995.

[4] J.A. Bergstra and J.-W. Klop. Process algebra for synchronous communication. *Information and Control*, 60 (1/3):109–137, 1984.

[5] J.A. Bergstra and A. Ponse. Process algebra with five-valued conditions. In C.S. Calude and M.J. Dinneen, editors, *Combinatorics, Complexity, and Logic, Proceedings of DMTCS'99 and CATS'99.* Springer-Verlag, Singapore, 1999. To appear.

[6] J.A. Bergstra and J.V. Tucker. Top-down design and the algebra of communicating processes. *Science of Computer Programming*, 5 (2):171–199, 1984.

[7] J.A. Bergstra and S.F.M. van Vlijmen. *Theoretische Software-Engineering.* ZENO-Institute, Leiden, Utrecht, The Netherlands, 1998. In Dutch.

[8] J.W. de Bakker. *Mathematical theory of program correctness.* Prentice Hall International, 1980.

[9] P. van Emde Boas. A compositional semantics for the Turing machine. In *J.W. de Bakker, 25 jaar semantiek, liber amicorum*, pages 219–227. CWI, Amsterdam, 1989.

[10] J.F. Groote and A. Ponse. Process algebra with guards. Combining Hoare logic and process algebra. *Formal Aspects of Computing*, 6(2):115–164, 1994. An extended abstract appeared in *Proceedings CONCUR 91,* Amsterdam, ed. J.C.M. Baeten and J.F. Groote, Lecture Notes in Computer Science, Vol. 527, 235–249. Springer-Verlag, 1991.

[11] C.A.R Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30 (8):672–686, 1987.

[12] J.-W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, volume II, pages 1–116. Oxford University Press, 1992.

[13] C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addisson-Wesley, The ACM-Press, New York, 1997.

A Appendix 1: program algebra embeddings

The program algebra projections translate programs in the direction of PGLA. In that direction program notations become less and less flexible. There is another line of operators which translates programs in the opposite direction. Such operations will be called program algebra embeddings because these explain the meaning of programs in terms of more flexible program notations. This paper gives rise to three such embeddings: the embedding of PGLA into PGLB, the embedding of PGLB into PGLC and the embedding of PGLC into PGLD.

A.1 Embedding PGLA into PGLB

The program algebra embedding `pgla2pglb` is obtained as follows. Given PGLA program X, one may first bring it into a second canonical form. Then there are three options: Y, Z^ω , $Y;Z^\omega$, in which Y and Z do not allow repetition. In the first case we define

$$\texttt{pgla2pglb}(X) = Y.$$

In the second case we write $X = Z;Z^\omega$, thereby reducing the second case to the third case. In the third case we write $X = u_1;...;u_k;(u_{k+1};...;u_{k+n})^\omega$ and we can define:

$$\texttt{pgla2pglb}(X) = u_1;...;u_k;\vartheta_1(u_{k+1});...;\vartheta_n(u_{k+n});\backslash\#n;\backslash\#n$$

The auxiliary operators ϑ_j are given by:

$$\vartheta_j(\#l) = \backslash\#n - l \ \texttt{if} \ j + l > n,$$

$$\vartheta_j(u) \ = u \ \texttt{otherwise}.$$

A.2 Embedding PGLB into PGLC

The program algebra embedding pglb2pglc is obtained as follows:

$$\texttt{pglb2pglc}(u_1;...;u_k) = \vartheta_1(u_1);...;\vartheta_k(u_k);\#;\#$$

In this case the auxiliary operators ϑ_j are given by:

$$\vartheta_j(\#l) = \#\ \texttt{if}\ j+l > k,$$

$$\vartheta_j(\backslash\#l) = \#\ \texttt{if}\ l \geq j,$$

$$\vartheta_j(!) = \#k + 3 - j,$$

$$\vartheta_j(u) = u\ \texttt{otherwise}.$$

A.3 Embedding PGLC into PGLD

Finally the program algebra embedding pglc2pgld is defined by:

$$\texttt{pglc2pgld}(u_1;...;u_k) = \vartheta_1(u_1);...;\vartheta_k(u_k)$$

In this third case the auxiliary operators ϑ_j are given by:

$$\vartheta_j(\#l) = \#\#j + l,$$

$$\vartheta_j(\backslash\#l) = \#\ \texttt{if}\ l \geq j,$$

$$\vartheta_j(\backslash\#l) = \#\#j - l\ \texttt{if}\ l < j,$$

$$\vartheta_j(u) = u\ \texttt{otherwise}.$$

B Appendix 2: the unit instruction operator

In this appendix we will briefly describe an operator which is immediately suggested by the setting of PGA, but which is not so standard. The unit instruction operator $u(-)$ takes a PGA program and wraps it into a unit which is taken to have length 1. The length matters, of course, in connection with the evaluation of the effect of jumps and tests. There are no useful program object equations for $u(-)$ known to us. We denote with PGLA_u the program notation that extends PGLA with the unit instruction mechanism. As it turns out the behavioral semantics of PGLA_u can easily be given by adding two program behavior equations to the ones given for PGA. We will list these equations below, after having made some introductory remarks. Following the philosophy of our main text, the semantics of PGLA_u should preferably be provided via a program algebra projection. Such a projection is feasible, and provides far greater flexibility than the direct behavioral semantics that we will offer in this appendix. The path to a program algebra projection for PGLA_u takes several steps. In a companion document we have outlined several extensions of PGLD, notably extensions that offer labels and goto's, as well as localization of the scope of

labels. At the highest level of these languages all dependence on instruction counting is removed, and the unit operator is semantically superfluous. Now one may translate PGLA_u into languages in this line of extension and at some stage simply delete the unit operators. Eventually a series of projection operators brings one back to PGLA.

The two additional program behavior equations dealing with the unit instruction operator are these.

$$|u(X)| = |X|$$

$$|u(X);Y| = |X;Y|$$

When reading the extended collection of equations it should be noticed that u will now range over program objects with length 1, which includes all programs of the form $u(X)$. That convention will hold as well for the existing program behavior equations, which are re-used here without being repeated.

Program Algebra with Comment Instructions

J.A. Bergstra* & M.E. Loots†

February 9, 1999

Abstract

The program notation PGLD is extended with comment instructions, thus obtaining the language extension PGLDc. A particular form of comment, the instruction count comments are shown to be helpful for both the design and the readable presentation of PGLD programs.

1 Introduction

The development of program algebra in [1] leads to four programming notations: PGLA, PGLB, PGLC, and PGLD. PGLA is minimal and conceptually orthogonal. It starts from the assumption that a program is a linear list of instructions and that an instuction can be removed from the list after having been processed. Termination requires an explicit instruction (!), and only forward jumps are possible, backward jumps not even making sense. Repetition is used to deal with programs that may perform an unbounded number of computation steps. Further, there is an abort instruction which allows problematic termination. In PGLB, the representation of programs takes the form of finite lists of instructions. Instead of repetition constructing infinite instruction lists, in PGLB the possibility to perform backward jumps is introduced in addition to the forward jumps inherited from PGLA. In PGLC the termination instruction disappears and, more conventionally, termination is obtained when a program has executed its final instruction, or when it performs a jump outside the range of used instruction numbers. Then in PGLD both forward and backward relative jumps are dropped in favor of absolute jumps.

The language PGLD is easy to explain informally: point of departure is a collection Σ of so-called atomic actions or atomic instructions. When performed all atomic actions return a boolean value. We list the basic constructions. For each atomic action $a \in A$ there is a program a which counts exactly one instruction. Then $\#$ is an instruction that represents abort. Further, for each natural number k, the instruction $\#\#k$ represents a jump to the start of instruction number k. The instructions $+a$ and $-a$ are called test instructions;

*University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl.

†Tilburg University, Faculty of Philosophy, email: M.E.Loots@kub.nl.

$+a$ and $-a$ are also single instruction programs. The test instructions work as follows: $+a$ is executed by performing a, which will deliver a boolean value, in addition to a potential effect on a state. If `true` is delivered, program execution will proceed according to normal flow of control. That means that if the test is the final instruction of the program, the execution terminates; in other cases, execution proceeds with the instruction following the test and so on. In case `false` is returned, execution terminates if the test is the final action, and proceeds after the first instruction following the test if it is not. So the effect of the boolean returned after a test instruction is to decide whether or not to execute the instruction just following the test action. In case of a negative test, the working is as for positive tests with the understanding that the returned boolean is negated before being handed over for inspection to program control.

1.1 Human readers and writers

The rationale of the program algebra language family as outlined above is primarily to provide program notations that are as simple as possible, to provide a very clear and concise behavioral semantics (as was done in [1]) and to have a platform for the understanding of assembly-level-programming and code generation. No attention has been paid to optimizing or even evaluating these formats as carriers for programs in a form that can be read, understood or even written by human agents. The languages PGLA-D have several drawbacks from a human point of view.

As a consequence it is not entirely clear where in the hierachy of languages built on top of PGLA, the first or simplest notation is to be found that can be used by a programmer (who must be a human being according to [2]). This is a pragmatic question. We have concluded that programming directly in PGLA is virtually impossible because the bookkeeping of the jumps will soon be too complex. The same holds for PGLB and for PGLC, albeit to a slightly lesser extent. In both cases it is too hard to get the jump counters right. In particular it is unclear how to make a stepwise development in which jump counters can be adapted. The solution is found in changing the language PGLD to PGLDg by adding the feature of gotos (and removing the then obsolete absolute jump instructions). PGLDg is suitable for human usage in simple circumstances. However, gotos and labels are not entirely straightforward because of alpha-conversion for labels, which introduces a non-trivial level of abstraction. The difficulty in writing PGLD programs, however, is that it is hard to get the jump counters correct at the time of writing. The difficulty in reading such programs lies in the fact that absolute addresses must be converted into line numbers, which will nearly always require computer support.

2 Comment instructions for PGLD

We will propose comment instructions. The language PGLD extended with the comment instruction option is called PGLDc. These comments will help a

human reader of a PGLDc program. The syntax of comment instructions is as follows: % is an empty comment instruction and for a string s that does not contain ';' % s is the comment instruction that gives comment 's'.

The relevance of comment instructions is to provide a human reader with information that is immaterial for the operational meaning of the program. For people to understand the meaning of the program, a projection function is needed which translates programs from the extended language PGLGc back to PGLD. This is the function `pgldc2pgld`, which works as follows:

$$\texttt{pgldc2pgld}(u_1; \cdots; u_k) = \psi_1(u_1); \cdots; \psi_k(u_k)).$$

The auxiliary operationss ψ_i read as follows:

$$\psi_i(\%) = \#\#i + 1$$

$$\psi_i(\%\ s) = \#\#i + 1$$

$$\psi_i(u) = u \ \texttt{otherwise}$$

If instruction number i equals $\#\#i + 1$, its effect upon execution is to pass control to the next instruction without any further side effect. The behavior of a PGLDc program X can be defined as $|X|_{pgldc} = |\texttt{pgldc2pgla}(X)|_{pgld}$. The behavior $|Y|_{pgld}$ of a PGLD program Y has been defined precisely in [1].

Comments are a tool for readers, which does not necessarily make them a tool for authors as well. In the case of PGLDc we will outline a kind of comment, the instruction count comment, which helps the reader much more than the author. In fact, the author is still faced with the original problem of having to compute the right counter of a jump while programming. The comment mentioned is as follows: '% `ic` k' is an instruction which asserts 'this is instruction number k'. A PGLDc program is instruction count comment correct (ICCC) if all instruction count comments assert the right thing about their position in the program. So $a; b; \%\texttt{lc}\ 3; +b; \#\#3$ is ICCC, whereas $a; b; c; \%\texttt{lc}\ 3; +b; \#\#3$ is not ICCC. An author who presents a program to a reader will try to ensure that it is ICCC. If not, the reader will feel free not to examine the program in further detail. The test is easy, but boring. The advantage of ICCC PGLDc programs is simply that a reader need not calculate what the jumps point to. In fact, the jump counters are like labels, albeit not in a mnemonic style. Although one can do without, the ease provided by mnemonic names is considerable.

The following example serves to illustrate the burden which is put on the reader by the use of jump counters, without instruction counter comments.

$$a; b; -g; \#\#21; b; b; \#\#14; +d; -e; +g; \#; \#; \#; \%\texttt{ic}\ 14; a; a; a; b; b; b; \%\texttt{ic}\ 21; f$$

which has the same behavior as

$$a; b; -g; \#\#20; b; b; \#\#14; +d; -e; +g; \#; \#; \#; a; a; a; b; b; b; f.$$

We consider the overhead in program length to be justified by the additional information concerning the targets of the jumps.

2.1 PGLDct, a design language for PGLDc

An author who is satisfied with PGLDc programs as textual products to be presented to his/her audience still faces the problem of having to write these programs. Now it is quite likely that the author knows how to proceed if the use of labels were allowed. The difficulty is that at the time of writing down an absolute jump instruction (in particular one that actually is a forward jump) it may not exactly be known to which instruction the jump has to move control. The standard solution to such matters is to take advantage of the philosophy of top-down design, and to use a template in preparation of the construction of the program. Templates stand for the real thing, which will be substituted when more information is available. In the case of PGLDc two template instructions are plausible: '$\#\#x$' and '$\%\texttt{ic}\ x$' where x is a variable for natural numbers. By allowing these extra instructions the program design notation PGLDct is obtained.

A design that can precede the example program above then is:

$$a; b; -g; \#\#x1; b; b; \#\#x2; +d; -e; +g; \#; \#; \#; \%\texttt{ic}\ x2; a; a; a; b; b; b; \%\texttt{ic}\ x1; f.$$

After having made this design, the author of the program will subsequently compute $x1 = 21$ and $x2 = 14$, and later substitute these values. In fact the design may be even easier to read than its translation into PGLDc, but there is a simple reason not to use these designs in some occasions. The cost of explaining the additional complexity of the variable binding may be prohibitive. As it is in general to be expected that for every program notation, the best design notation for it is a proper extension, one is not tempted to take a design notation as a means of presentation. Admittedly the cost of explaining a particular design in PGLDct is not very high: each variable must be instantiated with the (unique) instruction number of an instruction that features the variable in an instruction count comment. This can be formalized in the definition of a transformation `pgldct2pgldc`. The requirements of ICCC are more severe for PGLDct expressions. Not only should the instantiated instruction counts be correct, the template variables should all have a unique defining occurrence in a comment instruction. The test on LCCC and the transformation of a design expression to a PGLDc program can well be supported by an automated programming environment. The design language itself can also be viewed as a program notation, but the cost of its introduction may be considered prohibitive in some cases.

2.2 Using the design notation

Having the design notation available, the programmer may decide to first write a design and subsequently transform it into a program. In the case of PGLDc this strategy may well work for programs of 100 instructions or less. For much longer programs the readability of PGLDc is doubtful and the transformation from a design expression (written in PGLDct) which makes use of well-chosen mnemonic jump counter names, to an expression that uses instantiated jump

counters may be unreasonable. The problems appearing when having to present programs of increasing size explain many of the features that have been developed for modern programming notations. In principle there is an unbounded supply of possibilities to find short presentations of (previously long) programs.

What is clear from the above considerations is that the aspect of human readability, the use of comments and the distinctions of languages for engineering (design), presentation (with comments) and execution (without comments) is equally relevant for extremely simple languages. Some programming languages seem to be more complicated than necessary because an attempt has been made to include the features needed for design in the language itself. It can be expected that the more complex a languages becomes, the more discrepancies between syntactically correct programs and intermediate design expressions will appear.

3 Other comments

Of course, more comments are conceivable. For instance, in PGLDc it may be useful to have comments that simply provide a name for a section in a program or the date of production, the name of the author or any other useful kind of information. The name of a program can also be expressed in a comment. A systematic naming of such comments can be developed, but it will not be a concern in program algebra.

In other program notations based on program algebra, comments can be introduced in a similar way, provided that the projection removing the comments makes use of an instruction that has no side-effect and passes control to the next one. This instruction depends on the program notation used. For instance in PGLA, PGLB and PGLC it is the forward jump with counter 1: #1, and in a language with gotos it may be the catch of a label that is not used.

3.1 Concluding remarks

The decision to see comments as instructions is needed if programs are to be instruction lists and commented programs are to be viewed as programs as well. This is plausible because an explosion of the number of conceptual categories is not useful. For the same reason it is plausible to treat the design notation PGLDct for PGLDc simply as another program notation and to regard carrying out the transformation `pgldct2pgldc` as a task for the programmer (in some cases). As long as such a transformation can be easily automated there is no need to view design notations as a conceptually different category. A different situation arises if the transformation from design to program is not computable or clearly requires human intervention because of experience-based steps that one has not been able to capture in a more formal way.

References

[1] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998.

[2] J.A. Bergstra and S.F.M. van Vlijmen. *Theoretische Software-Engineering.* ZENO-Institute, Leiden, Utrecht, The Netherlands, 1998. In Dutch.

Projection Semantics for Flow Charts

J.A. Bergstra* & M.E. Loots†

March 28, 1999

Abstract

On the basis of the program notation PGLD of program algebra, a program notation PGLDg (PGLD with goto's) is proposed using indirect absolute jumps. An indirect absolute jump, also called a goto instruction, instructs the agent executing a program to move to an occurrence of a label catch instruction. The semantics of PGLDg is determined by a projection back to PGLD. PGLDg is an excellent point of departure for language extensions. As an example a language extension with a conditional construct is discussed. More importantly PGLDg is precisely adequate for the linearized representation of flow charts. Given the methodological importance of flow charts for programming, this fact marks PGLDg as the simplest notation in the program algebra hierachy for which there is a well-developed theory and methodology of software design.

1 Introduction

The point of departure for this paper is the program notation PGLD which was described in our [2]. Program algebra (PGA) as outlined there is an algebraic syntax for a very simple kind of programs. Closed terms over the syntax of PGA can be viewed as programs, and are called PGLA. PGLA involves atomic instructions, positive and negative tests, explicit successful termination, abort (unsuccessful termination), concatenation, repetition and forward jumps. The meaning of PGLA programs is defined by means of a mapping $|-|_{pgla}$ which assigns to a program expression X a behavior $|X|_{pgla}$.

We view PGLA as a manifest program notation in the sense of [5]. In doing so we take it for granted that the PGLA programs comprise lists of instructions amenable to execution by a computer, needing no further argument either from experience, from experiment, from convention or from convenience.

Our view on programming, at least on how the subject can be introduced from first principles, is very similar to the ideas proposed in connection with the toy robot programming system KAREL (and its OO enhancement KAREL++)

*University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl.

†Tilburg University, Faculty of Philosophy, email: M.E.Loots@kub.nl.

from [1]. Programming for KAREL starts from the same primitives as those introduced in the program algebra PGA. These primitives are restricted to actions, tests and control commands. Programming for KAREL may be seen as an instance of programming in a structured version of one of the PGA-based languages. Because the structured control features used in [1] can easily be formulated via flow charts, we aim at a very similar expressive power.

The purpose of this paper is to extend the development of program notations based on PGA with features that allow the linearized representation of flow charts. In doing so, a program notation PGLDg is developed that is significant in its own right. For PGLDg a projection semantics as well as a direct operational semantics is given. By translating flow charts into PGLDg a projection semantics for flow charts is immediately obtained.

1.1 Three arguments in favor of PGLDg

As PGLDg is not a structured language some argument is needed to motivate its introduction. It naturally emerges if one intends to model flow charts in a program algebra setting. Flow charts being quite important for the methodology of programming this point is significant. Secondly, one may view PGLDg as a technique for encoding finite transition systems in the notational style of program algebra. to imagine any program notation for digital computers that significantly moves outside the realm of finite transition systems. The importance of finite transition systems being undisputed, this counts as an important rationale for PGLDg. It is hard Thirdly, it turns out that PGLDg admits many interesting extensions which can be translated back to PGLD. Although PGLDg may not be a helpful vehicle for programmers, PGLDg is a very useful tool for semantic work.

We discuss a rule for extending the language and an example of a PGLDg extension is given. The example consists of a projection semantics for additional features regarding conditional constructs.

2 Flow Charts

Flow charts constitute a classical graphical device for denoting manifest software. We will not discuss the history of flow charts at this point. However, we will make an exception for [6, 7] and [8], since they too add an algebraic flavor to the formalization of flow charts. In [3], a connection is made with a more operational algebra.

It is safe to state that the flow chart notation is quite old, be it that different notational styles have emerged. The well-known Petri nets may be viewed as a very mature form of flow charts. In the case of flow charts, one visualizes the flow of control through a computation roadmap, to be distinguished from the case of data flow systems where the flow of data is being visualized. A flow chart consists of a collection of cells and a collection of flows. A cell contains an action. Actions are taken from a collection Σ serving as a parameter. These

actions are boolean-returning in the sense that an environment executing the actions will return a boolean value to the cell calling the action. There are two kinds of cells: void cells and test cells. A void cell ignores the returned boolean value. A test cell uses the returned boolean to decide how to proceed.

A flow is a connection, either between two cells or between a cell and the external context. We take the collection of flows to be a subset of the natural numbers. In the flow collection of a flow chart, 0 will represent the entry flow or incoming flow, along which control initiates computation, and 1 will represent the exit flow or outgoing flow, along which control leaves the computation of the chart after its completion. All other flows are called internal. These flow charts correspond to entire programs or systems.[1]

For every cell there is one entry flow, given by the entry flow function $entry(-)$. For every void action node there is an exit flow given by the exit flow function $exit(-)$. For every test cell there is a positive exit flow given by the function $+exit(-)$ and a negative exit flow written $-exit(-)$. All entry flows and exit flows are taken from the collection of flows that is a parameter of the flow chart.

There is a fundamental constraint: for every flow f different from the incoming flow 0 there is exactly one cell k (be it void or boolean-returning) for which $entry(k)$ equals f.

Computation of a flow chart works as follows: start in the incoming flow 0. Set the current flow cf to 0. Repeat the following steps until the outgoing flow is eventually reached, upon which the computation terminates (and thereby performs the termination behavior $||!||$): from current flow cf selects the unique cell k such that $entry(k) = cf$; then if this cell is a void action cell, perform its action (i.e. perform atomic behavior $||a||$ with a the label of k) and proceed (i.e. set cf equal) to flow $exit(k)$; otherwise, if it is a test cell perform its action, and depending on whether `true` (positive) or `false` (negative) was returned, set cf to $+exit(k)$ or $-exit(k)$ respectively.

The representation of flow charts as programs can easily be done at the level of the program notation PGLDg that is introduced below. For that reason we view PGLDg as a manifest program notation. It is conceived in order to be able to represent flow charts, which are definitely manifest. PGLDg may be seen as a language that offers a bare minimum of features needed for the representation of flow charts as linear program texts. That linearization being an important aspect of programming per se, we conclude that these linearized flow charts as well as the underlying flow charts are themselves manifest software.

2.1 Graphical display of flow charts

Flow charts probably owe most of their fame to the attractive possibilities of visualization they offer. We will not draw any flow charts so as to avoid the question how to formalize a picture. That may lead us to 2-dimensional grammars

[1] We will first describe standard flow charts that have exactly one incoming and one outgoing flow. In more advanced models there may be a number of incoming flows as well as a number of outgoing flows.

or to handwaving. We maintain that a flow chart is best seen as a mathematical entity which can, but need not, be depicted in a drawing. In the process of drawing up a flow chart several important transformations are usually carried out, which we will briefly describe. We will then outline how a picture can be formalized and may give rise to a flow chart in the sense as we previously defined it.

First of all the void cells will appear as squares with their action label written inside. The difference between cells simply emerges from their different positions in the picture. Cells are traditionally depicted as non-overlapping icons. For void actions one often uses a square with the entry incoming from above and the exit outgoing below. For test actions various icons are popular. Thus the collection of cells is visualized as a collection of disjoint icons. Similarly, test cells are often depicted as triangles, again with their action label written inside. For the sake of simplicity we will assume, however, that the test cells are also depicted as squares. Then flows take the form of curved lines. The entry flow of a cell 'flows into' it from above. In the case of a void action cell, the unique exit flow leaves the square from below. In the case of test cells, two lines will leave the square from below, one labeled with '$+$' the other with '$-$'. The incoming flow enters the picture from above and leads into exactly one icon, the outgoing flow leaves the picure from below. All flows have a direction. Flows may join but cannot fork. So the outgoing lines from two icons may join and then become entry lines for another icon. The logical flows as they occurred in the formalized description of flow charts above can be recovered as sets of lines that join into the same entry or that join into the outgoing flow. The numbering of the flows in the picture is of course superfluous for the purpose of distinction. Indeed, different flows are separated by one or more cells. When formalizing a depicted flow chart some numbering of the flows must be introduced arbitrarily, before the cells are mapped onto a set. At that stage the cell classification (void action or test action), the cell labeling, the entry mapping and the various exit mappings can be determined.

3 Four extensions of PGLA; a brief survey

PGLB extends PGLA with backward jumps on top of the forward jumps of PGLA. However, PGLB leaves out repetition, which can be simulated by means of a backward jump. The semantics of PGLB is determined by a mapping, called a program algebra projection, `pglb2pgla` from PGLB to PGLA. The meaning of PGLB program X can be determined as $|X|_{pglb} = |\texttt{pglb2pgla}(X)|_{pgla}$. Following [2] a program notation can in general be defined as a pair of a collection of syntactic objects L and a mapping (program algebra projection) φ from L to PGLA.

PGLC is a modification of PGLB, which has more standard termination conventions and leaves out PGLB's explicit successful termination. The semantics of PGLC is given via a translation `pglc2pglb`, which maps a PGLC program X to a (by definition equivalent) PGLB program $\texttt{pglb2pgla}(X)$. The meaning

$|X|$ of X is given by: $|X| = |\texttt{pglb2pgla}(\texttt{pglc2pglb}(X))|$. Writing $f \circ g(X)$ for $f(g(X))$ we can say that the semantics of a PGLC program X is given by $\texttt{pglb2pgla} \circ \texttt{pglc2pglb}$. The program notation PGLC offers a very simple format for assembly-like programs with control jumps based on offsets.

PGLD is yet another modification of these languages and will be the point of departure for the present paper. It replaces the relative jumps of PGLC by absolute jumps. An absolute jumps takes as its parameter a natural number, which is interpreted as the program counter in the instruction list. The semantics is determined by a mapping `pgld2pglc`, which was outlined in [2].

3.1 Labels and goto's

Based on PGLD we will outline further language extensions (or modifications) which allow one to use labels. In PGLDg goto's and labels are introduced. A label is just a natural number. PGLDg omits the absolute jumps as those can be replaced by an appropriate use of labels. The methodological advantage of the use of labels is a matter of program flexibility. Local changes of the code may affect the instruction count but will show no side-effects on the connection between labels and goto's, provided confusion between labels is avoided.

4 Program Expression Syntax for PGLD

The program notation PGLD has been introduced in [2]. That paper also contains the description of a transformation (projection) which allows one to translate a PGLD program back to a PGLA program. Because we will need PGLD, we will focus on an introduction of that notation per se. The detour along the more elementary forms of program algebra being ignored, a behavioral semantics can be provided in a direct fashion.

The syntax of program expressions in PGLD is generated from five constants (or kinds of constants) and one composition mechanism. The constants can be viewed as basic instructions. The composition mechanism is the structuring 'feature' of the programming language: text-sequential composition of X and Y, written $X;Y$. The rationale of this syntax is based on the Program Algebra that we have outlined in [2]. As a parameter we need a collection Σ of so-called atomic actions. These actions are considered boolean-returning. This means that every execution of the actions produces a boolean value. That value is returned to the calling program in the case of test instructions and is omitted in the case of void instructions.

void atomic instruction. All elements $a \in \Sigma$ serve as atomic actions or instructions; when executed these actions may modify (have a side-effect on) a state. The attribute void indicates that a returned boolean value is not used by the program. After having performed an atomic instruction a program must enact its subsequent instruction. If that instruction fails to exist termination occurs.

positive test instruction. For all actions $a \in \Sigma$ there is the negative test instruction $+a$. If $+a$ is performed by a program, the state is affected according to a, whereafter the sequence of remaining actions is performed in case true was returned. If there are no remaining actions program execution will terminate. If false was returned after a was performed, the next action is skipped and execution proceeds with the instruction following that skipped instruction.

negative test instruction. For all actions $a \in \Sigma$ there is the negative test instruction $-a$. If $-a$ is performed by a program, the state is affected according to a, whereafter the sequence of remaining actions is performed in case false was returned. If the test is the last action of the program the execution will terminate after its execution. If true was returned after a was performed, the next action is skipped and execution proceeds with the instruction following that skipped instruction.

absolute jump instructions. For any natural number k in NN[2] there is an action $\#\#k$ which denotes a jump to the k-th instruction of the program. We call k the counter of the (absolute) jump instruction. If $k = n$, with n the position in the sequence of instructions at which the jump is placed, the jump is to the very instruction itself (zero steps forward). In this case a loop or divergence will result. If the counter of the jump equals $n+1$ the instruction is a skip (i.e. it skips itself). If $k = 0$ or k exceeds the number of instructions of its program then termination of the program execution will take place upon execution of that jump instructon.

abort instruction. The jump without counter (#) serves as a notation for an instruction resulting in the unsuccessful termination (abort) of the program.

Examples of PGLD programs (or rather program expressions) are:

- $a; b; c$
- $b; +a; \#\#5; \#\#2; -c; +c$
- $\#\#1$

4.1 An algebra of program objects for PGLD

The closed expressions in PGLD constitute an algebra of programs objects after dividing out the congruence generated by the associativity of concatenation. It follows that each program can be seen as a sequence of instructions: $X = u_1; ...; u_k$. As we have stated in [2] the program objects are the elements of a program algebra, whereas the semantic constructions that are assigned to these program objects need not constitute a program algebra. We will use behavioral abstraction as a semantics for the program objects. In principle the

[2]We use NN for the set of natural numbers and PNN for the set of positive natural numbers.

semantics of PGLD programs is to be obtained via the projection pgld2pgla as follows: $|X|_{pgld} = |\texttt{pgld2pgla}(X)|_{pgla}$. This equation expresses the underlying assumption that as far as semantics is concerned the projection pgld2pgla takes priority over other considerations.

We will not repeat the details of the program algebra projection pgld2pgla at this point, these matters having been documented in [2]. No familiarity with these transformations is needed for a proper understanding of the remainder of this paper. Besides the behavior assigned to a PGLD program X can be described directly in terms of a system of recursion equations associated with the structure of X. This will be done in the following paragraph. It can help the reader to obtain an intuition for the meaning of PGLD expressions. It should be noticed, however, that whenever another semantic assignment is used for PGLA, that semantic modification will automatically entail a modification in the semantics used for PGLD, projection having priority over any other semantic consideration.

A direct description of the behavior of programs in a program notation, following the intuitive understanding of its execution model, is called an operational semantics. Next we will present an operational semantics for PGLD.

4.2 Operational semantics for PGLD

For each closed program expression of PGLD a behavior can be defined. The preferred way to obtain this behavior is via a program algebra projection into PGLA following [2]. Here we will provide a direct description of that behavior. We will use an auxiliary operator $|-,-|_d$ and we have $|X|_{pgld} = |1, X|_d$. This way an independent description of the behavior of PGLD programs is obtained.

With $|i, X|_d$ we denote the behavior of X if its execution is started up with its i-th instruction. Here i can be an arbitrary natural number. Let $X = u_1; ...; u_k$. The semantic equations for $|-,-|_d$ are:

$$|j, X|_d = ||!|| \texttt{ if } j > k \texttt{ or } j < 1$$

$$|j, X|_d = ||a|| \cdot |j+1, X|_d \texttt{ if } u_j = a$$

$$|j, X|_d = ||a|| \cdot |j+1, X|_d \texttt{ if } u_j = +a \texttt{ and } Y(a) \texttt{ holds}$$

$$|j, X|_d = ||a|| \cdot |j+2, X|_d \texttt{ if } u_j = +a \texttt{ and } Y(a) \texttt{ is false}$$

$$|j, X|_d = ||a|| \cdot |j+1, X|_d \texttt{ if } u_j = -a \texttt{ and } Y(a) \texttt{ is false}$$

$$|j, X|_d = ||a|| \cdot |j+2, X|_d \texttt{ if } u_j = -a \texttt{ and } Y(a) \texttt{ holds}$$

$$|j, X|_d = M \texttt{ if } u_j = \#$$

$$|j, X|_d = |l, X|_d \texttt{ if } u_j = \#\#l$$

Here $||!||$ is the irreducible behavior of the successful termination instruction, $||a||$ denotes the primitive behavior that describes the execution of a single atomic action, and $Y(a)$ is a predicate that predicts whether action a will return `true` or `false`. Additionally, $P \cdot Q$ represents the sequential composition of P and

Q. If repeated application of these equations leads to no primitive behavior, the divergent behavior D is assigned as a meaning to a program. This happens only in case there is a loop in the program. M is the meaningless program which denotes the semantics of the abort instruction. For a theory of behaviors with D and M we refer to [4].

We notice that pairs of equations can be combined as follows, be it at cost of readability. The 3-th and the 4-th equation together are equivalent with:

$$|j, X|_d = (||a|| \cdot |j+1, X|_d) \lhd Y(a) \rhd (||a|| \cdot |j+2, X|_d) \ \texttt{if} \ u_j = +a$$

Similarly equations 5 and 6 can be combined into:

$$|j, X|_d = (||a|| \cdot |j+2, X|_d) \lhd Y(a) \rhd (||a|| \cdot |j+1, X|_d) \ \texttt{if} \ u_j = -a.$$

5 Labels and goto's in PGLDg

We will outline an important modification of the language PGLD. First the syntax is given, together with an intuitive account of the meaning of its control mechanisms. Secondly a program algebra projection into PGLD is provided (from which point onwards downstream transformations into PGLA are familiar). For PGLDg there is in fact a context-free language.[3]

5.1 Program expression syntax for PGLDg

As in PGLD, the only composition mechanism of programs in PGLDg is concatenation. PGLDg differs from PGLD in containing labels and goto's. Because they can be emulated by means of labels and goto's, absolute jumps have been deleted, the language becoming more homogeneous as a result. Incorporation in PGLDg of absolute jumps would not introduce any conceptual or semantic difficulties, however, and can under circumstances be quite practical.

As labels we will use the natural numbers. $£k$ denotes a label with name k with $\#\#£k$ a jump towards a label is denoted. Such jumps are also called goto's and for some reason they have a bad reputation.[4] The argument in favor of

[3] An expression X outside L can be viewed as an L-program with (static) errors. The compiler or syntax checker should reject the program. Having thus obtained a syntactical category of programs, a program algebra projection will determine their semantics. In many cases static errors can be replaced by dynamic ones. That works fine in theory. In practice, however, one better works the other way around: useless programs should be rejected by the syntax and type checking utilities of a compiler or more generally a programming environment. The argument is that whenever (easily) possible the need for dynamic tests should be minimized. So in the case of PGLDg we consider it better to reject programs with 'dangling labels' than to invoke meaningless behavior in case a goto into such a label is to be performed.

[4] There is a convincing literature about unfortunate use of labels and goto's. We fail to grasp, however, how such observations can substantially reflect on the value or even the perception of the value of such extremely clearcut mechanisms as labels and goto's. All that can be said is that labels and goto's may be too low level mechanisms for 'end programmer expression'. The same holds for binary strings and TCP/IP protocol frames and those have not (yet?) acquired a bad reputation.

labels and goto's coincides with the argument against absolute or relative jumps. Absolute jumps depend strongly on the sequential numbering of instructions. Every modification of a program that affects this numbering may have as a side-effect that many jump instructions have to be changed. This is very impractical for a programmer (and for an author of software transformation programs). Now the key observation is that what is really meant by an absolute jump is a jump to a position in a program.

For a programmer this position is mainly determined by a pattern of a limited number of instructions before and after it that together can be assigned a coherent meaning. So a logical naming scheme for such positions may be more useful. Indeed, one may view the instruction count as a naming of program positions by means of a program counter. A jump then accesses a position via its name, which happens to coincide with its sequence number in the program listing. What labels and goto's provide is a level of indirectness. A label may be viewed as a pointer to an instruction sequence number rather than as a number proper. Now local changes at one end of a program will in general not have any impact on logical positions at another end of the program and as a consequence labels and goto's introduce flexibility for a programmer (be it a hypothetical one). As soon as one views a program in terms of logical positions rather than instruction sequence numbers, the need for absolute jumps rapidly decreases and therefore, as mentioned above, those have not been incorporated in PGLDg. The effect of the introduction of labels on the collection of instructions is hardly significant. We outline the class of instructions below:

- Actions and tests like in PGLD. These comprise: void atomic instruction (a), positive test instruction ($+a$), negative test instruction($-a$) and the abort instruction ($\#$).
- Absolute goto instruction. For each non-zero natural number k the instruction $\#\#£k$ represents a jump to the (beginning of) the first instruction in the program which is labeled by the label k. If no such instruction can be found termination of the program execution will occur. (G-occurrence)
- Label catch instruction. The instruction $£k$, for k a natural number, represents a visible label. As an action it is a skip in the sense that it will not have any effects on a state space. (L-occurrence)

An example of a PGLDg program is:

$$£0; -a; \#\#£1; \#\#£0; £1$$

In this program a is repeated until it returns value `false`. That is also the functionality of the simpler program $£0; +a; \#\#£0$.

5.2 A projection from PGLDg to PGLD

We take it for granted that the reader has an intuitive grasp of the meaning of these programs. Nevertheless we will determine their meaning by providing

the details of a projection from PGLDg into PGLD. Given the familiar program algebra projection for PGLD (from [2]) a program algebra projection for PGLDg results. Using the direct behavioral semantics provided for PGLD above, the mapping from PGLDg to PGLD also brings about a behavioral semantics for PGLDg without any reference to [2]. This can be summarized with the following equation:

$$|X|_{pgldg} = |\texttt{pgldg2pgld}(X)|_{pgld}.$$

The projection `pgldg2pgld` from PGLDg to PGLD is quite simple. For a program X, the following steps have to be taken:

1. Replace for each label k, the goto instruction $\#\#\pounds k$ by $\#\#n$ with n equal to the smallest j such that the j-th instruction contains an L-occurrence of the label k, if that instruction exists, or replace it by $\#\#0$ otherwise.
2. Each label catch instruction $\pounds k$ is replaced by $\#\#j+1$ with j the instruction number of that label catch instruction. (This is a slightly cumbersome way to produce a skip instruction.)

More formally the projection `pgldg2pgld` works as follows on a program $u_1; ...; u_k$:

$$\texttt{pgldg2pgld}(u_1; ...; u_k) = \psi_1(u_1); ...; \psi_k(u_k)$$

where the auxiliary operation ψ_j reads as follws:

$$\psi_j(\#\#\pounds k) = \#\#\texttt{target}(k)$$

$$\psi_j(\pounds k) = \#\#j + 1$$

$$\psi_j(u) = u \ \texttt{otherwise}$$

The auxiliary function $\texttt{target}(k)$ produces for k the smallest number j such that the j-th instruction of the program is of the form $\pounds k$, if such a number exists and 0 otherwise.
Example: $\texttt{pgldg2pgld}(\pounds 0; +a; \#\#\pounds 1; \#\#\pounds 0; \pounds 1) = \ \#\#2; +a; \#\#5; \#\#1; \#\#6$.

5.3 Operational semantics for PGLDg

It is also possible to provide a direct description of the behavioral semantics on the basis of the syntax of PGLDg. It should be noticed that in case of doubt the projection semantics takes priority. This is important when a modification of the behavioral semantics used for PGLD is considered. Then automatically, via the projection semantics, the behavioral semantics of PGLDg is modified as well.

In view of the significance of PGLDg for program algebra it is useful to provide a direct description of the behavioral semantics of PGLDg programs in addition to the projection semantics that already given. We will use an auxiliary operator $|X, j|_g$, denoting the behavior of a program X when started just before its instruction number j. In this description $||a||^{\bullet}$represents the

behavior of the action a, and $||!||$ represents the termination of the system. M represents an abort (termination due to an error.) We have $|X|_{pgldg} = |1, X|_g$. Let $X = u_1; ...; u_k$. The semantic equations for $|-, -|_g$ are:

$$|j, X|_g = ||!|| \text{ if } j > k$$

$$|j, X|_g = ||a|| \cdot |j + 1, X|_g \text{ if } u_j = a$$

$$|j, X|_g = ||a|| \cdot |j + 1, X|_g \text{ if } u_j = +a \text{ and } Y(a) \text{ holds}$$

$$|j, X|_g = ||a|| \cdot |j + 2, X|_g \text{ if } u_j = +a \text{ and } Y(a) \text{ is false}$$

$$|j, X|_g = ||a|| \cdot |j + 1, X|_g \text{ if } u_j = -a \text{ and } Y(a) \text{ is false}$$

$$|j, X|_g = ||a|| \cdot |j + 2, X|_g \text{ if } u_j = -a \text{ and } Y(a) \text{ holds}$$

$$|j, X|_g = |j + 1, X|_g \text{ if } u_j = £k$$

$$|j, X|_g = M \text{ if } u_j = \#$$

$$|j, X|_g = |m, X|_g \text{ if } u_j = \#£l \text{ and } m > 0 \text{ with } m = \texttt{target}(l, X)$$

$$|j, X|_g = ||!|| \text{ if } u_j = \#£l \text{ and } m = 0 \text{ with } m = \texttt{target}(l, X)$$

The operator `target` takes a label number k and a PGLDg program and computes the lowest j such that $u_j = £k$ if that exits and 0 otherwise. If repeated application of these equations leads to no primitive behavior the divergent behavior D is assigned as a meaning to a program. This happens only in case there is a loop in the program, which involves no instruction from Σ. M is the meaningless program denoting the semantics of the abort instruction.

5.4 Embedding PGLD into PGLDg

The embedding `pgld2pgldg` of PGLD into PGLDg works as follows: let $X = u_1; ...; u_k$, then the embedded version is defined as

$$\texttt{pgld2pgldg}(u_1; ...; u_k) = £0; \vartheta_1(u_1); ...; \vartheta_k(u_k); £k + 1.$$

Here $\vartheta_j(u)$ reads as follows:

$$\vartheta_j(\#\#l) = £j; \#\#£k + 1 \text{ if } l \geq k + 1$$

$$\vartheta_j(\#\#l) = £j; \#\#£l \text{ if } l < k + 1$$

$$\vartheta_j(+a) = £j; +a; \#\#£j + 1; \#\#£j + 2$$

$$\vartheta_j(-a) = £j; -a; \#\#£j + 1; \#\#£j + 2$$

$$\vartheta_j(u) = £j; u \text{ otherwise.}$$

5.5 A rule for extending PGLDg

Extending PGLDg can be done by means of the introduction of additional actions outside the atomic action parameter set Σ. Such actions will be called *advanced control instructions*. Care should be taken not to use advanced control instructions immediately after a positive or negative test. The reason for imposing this restriction on extensions of PGLDg is that in most cases a semantic projection of the language extension will transform its advanced control instructions into sequences of PGLDg instructions (perhaps in a context-dependent way). This may then lead to a difficulty in the intuitive understanding of the test instruction. In case an advanced control instruction must be skipped: it is not clear whether it the entire translated sequence of actions or just their first element. To avoid this question the aforementioned restriction will suffice. As a goto instruction is not considered an advanced control instruction, the expressive power of the program notations is not influenced by this restriction.

6 Linearization of flow charts in PGLDg

The description of flow charts that was provided in paragraph 2 provides no immediate clue as to how to embed flow charts into a program notation. We will now indicate that PGLDg has exactly the adequate expressive power to do so. To each flow chart (in the formal sense) FC we will assign a program `ffcs2pgldg`(FC) in PGLDg which represents that flow chart and can be taken as a program notation for it. We consider the connection between the setting of formalized flow charts FFCS and PGLDg to be so close that it justifies extending the classification of FFCS as manifest software to PGLDg.[5] We will need a technical notion concerning the structure of programs in PGLDg. A normal control flow segment is a program in which normal flow of control can lead to an execution from the start to the end without any goto being executed (except the last action, which is allowed to be a goto). NFCS, the class of normal flow of control segments, is defined inductively as follows:

- a, $+a$, $-a$, $\pounds k$, and $\#\#\pounds k$ are in NFCS,
- if X is in NFCS then so are:
  1. $a;X$ for all atomic actions a,
  2. $\pounds k;X$ for all k,
  3. $+a;u;X$ and $-a;u;X$ for all atomic actions a and for all instructions u,
- No other PGLDg programs are in NFCS.

A program is in segmented form if it has the form $X_1;\#;X_2;\#;...;\#;X_n$ with each of the X_j in NFCS. For a given flow chart FC we will define a

[5]The mechanisms incorporated in PGLDg provide another argument for the classification of PGLDg as a language for manifest software.

program `ffcs2pgldg`(FC) in segmented form representing the flow chart FC. Let $C_1, ..., C_n$ be an enumeration of the cells of the flow chart. For every cell C_j the program P_j is as follows:

- if C_j is a void action cell with label a and $entry(C_j) = f, exit(C_j) = g$ then $P_j = £f; a; \#\#£g$,
- if C_j is a test action cell with label a and $entry(C_j) = f, +exit(C_j) = g, -exit(C_j) = h$ then: if $g = h$ we take $P_j = £f; a; \#\#£g$ and if $g \neq h$ we write $P_j = £f; +a; \#\#£g; \#\#£h$.

Having these programs available per cell, we find:

$$\texttt{ffcs2pgldg}(FC) = \#\#£0; \#; P_1; \#; P_2; \#; ...; \#; P_n; \#; £1.$$

We recall that the entire flow chart will have 0 as its incoming and 1 as its only outgoing flow.

In most cases it will be possible to find a shorter representation of a flow chart using larger segments. Such a representation will use normal flow of control to move from the code of one cell to that of another one. The more efficient representation will be behaviorally equivalent to the given one. We will not pursue that matter any further here as it seems to be of little practical interest. Most likely a compiler optimization utility would be able to generate the transformation to more compact code for a flow chart from the given representation.

7 Projection semantics for conditional constructs

On top of PGLDg it is easy and for the job of denoting programs clearly helpful to offer the conventional conditional constructs. These can be viewed as design patterns from the point of view of PGLDg programming. We will now provide a syntax PGLDgc allowing conditional instructions. Then a projection is given from PGLDgc into PGLDg.

7.1 Syntax for conditional instructions

The syntax for conditional constructs takes the form of three new (advanced control) actions (see paragraph 5.5):

if-instruction. For an action $a \in \Sigma$ the instructions `if` $+a\{$ and `if` $-a\{$ initiate the text of a conditional construct.

then/else separator. The instruction '}{' connects two program sections that are enclosed in braces.

end brace. The instruction '}' serves as a closing brace in connection with its complementary opening brace.

An example clarifies the intended meaning of (this rendering of) conditional instructions. The subsequent program algebra projection into PGLDg formalizes the same. The program

$$\texttt{if} + a\{; b; +c; \#\#£0; \#\#£1; \}\{; e; f; £0; g; £1; h; \}; -a; \#; b$$

will start with the execution of a. If that yields `true` the first branch is taken. Execution of the first branch will unavoidably lead to a jump into the second branch. It depends on the yield of c at which position in the second branch the computation proceeds. If the yield of the action a mentioned before has been `false` then the second branch (beginning just after the separator instruction '}{') is executed.

7.2 A projection operator

The mapping `pgldgc2pgldg` determines the semantics of these constructs. It simultaneously removes all occurrences of `if-{;-;}` and of `if-{;-;}{;-;}`. All labels that occur in the program before transformation are multiplied by 3. This will guarantee that labels of the form $3 \cdot i + 1$ and $3 \cdot i + 2$ are new and cannot interfere with existing labels. Let $X = u_1; ...; u_k$. Then we read:

$$\texttt{pgldgc2pgldg}(X) = \psi_1(u_1); ...; \psi_k(u_k)$$

with the auxiliary operations ψ_i determined by the following rewrite rules (with positive conditions, negative conditions and priorities):

$$\psi_i(\#\#£l) = \#\#£3 \cdot l$$

$$\psi_i(£l) = £3 \cdot l$$

$$\psi_i(\texttt{if} + a\{) = -a; \#\#£3 \cdot i + 1$$

$$\psi_i(\texttt{if} - a\{) = +a; \#\#£3 \cdot i + 1$$

$$\psi_i(\}) = \#\ \texttt{if coind}(i) = 0$$

$$\psi_i(\}) = £3 \cdot j + 1\ \texttt{if}\ j = \texttt{coind}(i) \neq 0\ \texttt{and}\ \texttt{kind}(j) = \texttt{single}$$

$$\psi_i(\}) = £3 \cdot j + 2\ \texttt{if}\ j = \texttt{coind}(i) \neq 0\ \texttt{and}\ \texttt{kind}(j) = \texttt{double}$$

$$\psi_i(\}\{) = \#\#£3 \cdot j + 2; £3 \cdot j + 1\ \texttt{if}\ j = \texttt{coind}(i) \neq 0\ \texttt{and}\ \texttt{kind}(j) = \texttt{double}$$

$$\psi_i(u) = u\ \texttt{otherwise}$$

The auxiliary function `coind` has type {1,...,k} → {0,1,...,k}. `coind`(i) equals 0 if u_i is not '}' or '}{', or if u_i = '}' and u_i is not the closing brace of an if-instruction, or of u_i = '}{' and u_i is not the intermediate separator of an if-instruction. Otherwise `coind`(j) equals the index of the if-instruction of which it is a closing brace or an intermediate separator. The auxiliary function `kind` has domain {1,...,k} and codomain {`single`,`double`,`noif`}. `kind`(i) = `single` if u_i is an if-instruction that begins a conditional construct without separator, and it equals `double` if u_i begins an if-statment with a separator instruction inside; otherwise `kind`(i) = `noif`.

7.3 Compile time data left unspecified

We will not formally specify the auxiliary operations that have been used in the description of the projection `pgldgc2pgldg` in more detail. For instance computing `coind` is just a matter of finding the corresponding opening brace ignoring the separator instructions. In order to determine `kind`(i) one views the separators as brace pairs, and starts looking for the corresponding closing brace for the opening brace in the if-instruction. If it is a solitary closing brace, the if-instruction (or rather its index) is single (armed), if it happens to be a brace included in a separator instruction the if-instruction is double (armed) and in all other cases the index is 'noif'.

It should be noticed that the information contained in the operators just mentioned must be discovered and used at compile time. It will not play a role during the run time execution of the translated program.

8 Relative indirect jumps

The notation that we have used has been designed in such a way that some interesting options are still open. In particular we will use this appendix to indicate how a very precise match with the developments in [2] can be obtained. In [2] we have outlined four program notations: PGLA, PGLB, PGLC and PGLD. PGLA and PGLB require that an explicit termination action (denoted with !) is executed in order for a program execution to properly terminate. Running out of instructions leads to an error. In contrast PGLC and PGLD do away with an explicit termination instruction and have program execution terminate whenever the last action of a program has been performed, or if a jump is made into a non-existing instruction in the program. A second difference between these languages is that PGLA allows forward jumps only, but has a repetition construct. PGLB and PGLC have no repetition construct but allow forward as well as backward jumps. PGLD has no repetition construct either, its jumps being absolute (counting instructions from the beginning of the program) rather than relative (as in PGLA, PGLB and PGLC).

The goto's instructions of the form $\#\#\pounds k$, which have been introduced and used in the present paper, are absolute in the sense that the corresponding label catch instruction is to be found by carrying out a search from the beginning of the program. So PGLDg is a goto/label version of PGLD. Instead of PGLD's absolute jumps PGLDg allows absolute goto's.

With $\#\pounds k$ we denote a forward goto instruction. When a forward goto is executed a corresponding label catch is searched for from the instruction containing the goto onwards. With $\backslash\#\pounds k$ we indicate a backward goto instruction. To execute a backward goto a matching label catch is searched by looking backward in the program text from the instruction containing the backward goto. Both forward and backward goto's are relative goto's rather than absolute ones.

8.1 Three more program notations

With PGLAg we denote the variation of PGLA obtained by allowing forward goto's rather than forward jumps. In this case termination requires the special action, and a forward goto for which no catching label instruction is found results in an error (#).

With PGLBg we will denote the version of PGLB that is obtained if forward and backward goto's are allowed instead of forward and backward jumps. The termination conventions are the same as in PGLAg. A backward goto that cannot be caught also leads to an error.

With PGLCg we denote the version of PGLG in which forward goto's replace the forward jumps and backward goto's replace the backward jumps. The termination conventions are now different: a goto without a corresponding catch leads to proper termination.

PGLDg has already been introduced in exactly the same fashion. Evidently a matrix of projections and embeddings between these program notations can be designed. We will not provide that information in this paper, however.

References

[1] J. Bergin, M. Stehlik, J. Roberts, and R. Pattis. *Karel++, a gentle introduction to the art of object-oriented programming.* John Wiley and Sons, 1997.

[2] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998.

[3] J.A. Bergstra, C.A. Middelburg, and Gh. Ştefănescu. Network algebra for asynchronous dataflow. *International Journal of Computer Mathematics*, 65:57–88, 1997.

[4] J.A. Bergstra and A. Ponse. Process algebra with five-valued conditions. In C.S. Calude and M.J. Dinneen, editors, *Combinatorics, Complexity, and Logic, Proceedings of DMTCS'99 and CATS'99.* Springer-Verlag, Singapore, 1999. To appear.

[5] J.A. Bergstra and S.F.M. van Vlijmen. *Theoretische Software-Engineering.* ZENO-Institute, Leiden, Utrecht, The Netherlands, 1998. In Dutch.

[6] V.E. Căzănescu and Gh. Ştefănescu. A formal representation of flowchart schemes I. *Analele Universitătii Bucuresti, Matematică - Informatică*, 37:33–51, 1988.

[7] V.E. Căzănescu and Gh. Ştefănescu. A formal representation of flowchart schemes II. *Studii si Cercetări Metematice*, 41:151–167, 1989.

[8] Gh. Ştefănescu. Algebra of flownomials. Part 1: Binary flownomials, basic theory. Technical Report I9437, Department of Computer Science, Technical University Munich, October 1994.

Abstract Data Type Coprograms

J.A. Bergstra* & M.E. Loots†

March 30, 1999

Abstract

Coprograms consume and process actions complementary to programs issuing actions. Coprograms maintain a state and reply with a boolean response to each action in their interface. As coprograms offer a service to other units, their interface is also called a service interface. Coprograms can be combined with several natural operators, thus giving rise to a coprogram calculus. Abstract data type coprograms are used for abstract data type modeling.

1 Introduction

Coprograms are used to represent data structures and data types in a context focussing on their role within programs. Data types are given a secondary role. Although programs are given the lead and data types are considered auxiliary in nature, the formalization of data types is still of major importance for program understanding.

Data type modeling requires as much care as the analysis of programs and programming concepts. Coprograms admit several interesting operators, so there is an algebra of coprograms as well as there is an algebra of programs.

We prefer to call it a calculus of coprograms.[1] We will provide an informal introduction to the operational intuition of coprograms. Subsequently we discuss a pedestrian classification of specification techniques for coprograms, followed by an equally informal classification of interfaces. The calculus of coprograms is discussed in some detail and examples are provided.

2 Coprograms

A coprogram is a pair consisting of an interface and a reply function. The interface in turn consists of a collection of actions (also called instructions), the

*University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl

†Tilburg University, Faculty of Philosophy, email: M.E.Loots@kub.nl.

[1]The reason for this choice is that identity in the program calculus is defined in terms of set-theoretic constructions rather than directly or indirectly by means of equational reasoning.

reply function being a mapping that gives for each finite sequence of interface actions (i.e. actions from the interface) the reply produced by the coprogram. This reply is a boolean value (`true` or `false` unless an error has occurred (M) or divergence occurs (symbolized by D).

2.1 Operational intuition of coprograms

The intuition is as follows: an agent outside the coprogram makes use of the service offered by the coprogram. These services are listed as instructions in the service interface of the coprogram. Each instruction in this interface can be processed by the coprogram. The coprogram updates its state and produces a reply in the form of a boolean value which is returned to the agent that invoked the service. Two errors can occur: abort (M) or divergence (D). In both cases the coprogram replies with a corresponding error value rather than an ordinary boolean. Further, it will not process any subsequent requests after having run into an error. It is assumed that the agent starts the operation of the coprogram with its first request for a service. The coprogram is always initialized from the same initial state.

2.1.1 The role of coprograms

Coprograms emerge first of all as a tool for the description of data structures. Well-known data structures such as memory registers, stacks, queues and tables can all be modeled as coprograms. Although the terminology of coprograms seems to be novel, the concept of coprograms is not new.[2] If one forgets the use of divergence (D) and error (M), it turns out that a coprogram is simply a formal language, perhaps based upon an unusual alphabet of symbols.

2.1.2 Abstract data type coprograms

Abstract data coprograms describe abstract data types in such a way that they can be used as part of an abstract machine. There is no implication that the abstract data type, or its coprogram must be realized either in hardware or in software if it is used in a system description. Its role is restricted to specification. In many cases specifications involving abstract data type coprograms can be transformed into implementations in a fully automatic and an algorithmically satisfactory way.[3]

[2] In [1] they appear under the name of processes, a phrase that is currently used for a far more general kind of behavior. We refer to [2] for a discussion of abstract data type behavior motivating the role coprograms in this paper. An extensive literature on observability in data types relates to very similar notions.

[3] The transformation used for that purpose, however, need not generate a system that has the same or virtually the same architecture as the given specification. Only the externally visible behavior must comply with the specification, all of its internal details in fact being private to the specification.

2.1.3 Program components and coprogram components

A program component[4] is a pair of an interface Σ and a program such that all basic instructions used by the program are in Σ. This definition is meaningful for all program notations in program algebra. A coprogram has a definite interface and coprogram components will be identified with coprograms.

We propopose the phrase i-program (interfaced program) for a program component. We write X^i for interfaced programs. The interface is extracted by an application of the operator Σ_l to X^i.

2.2 Coprogram details

A coprogram H is a pair (Σ, F), where Σ is a collection of actions constituting the service interface of H, and F is the so-called reply function. F takes as arguments the non-empty sequences of actions in Σ and has outputs in $\{\mathtt{true}, \mathtt{false}, D, M\}$. The requirement is that for $a \in \Sigma$ and $w \in \Sigma^*$ if $F(w) = D$ then also $F(w, a) = D$ and if $F(w) = M$ then also $F(w, a) = M$. Here w, a indicates the extension of list w with a and Σ^* denotes as usual the family of (empty and non-empty sequences) of elements of Σ. We will use the convention that sequences of interface actions are separated by commas.

For a set Σ the collection of all coprograms with service interface Σ is denoted with $\mathtt{Coprog}(\Sigma)$. This collection can be considered the semantics of the service interface Σ. We will not discuss the set-theoretic size of this collection. It is quite large but only a fraction of its conceivable elements has some relevance for computer programming.

3 System components and polarized interfaces

We will use system component to denote any kind of component. The phrase software component, unfortunately, has become charged with a technical meaning, thus making it less useful for general purposes. The phrase system component should be protected from being given a specific technical meaning.

3.1 Interfaces and polarization

An interface is a set of (names of) instructions (or actions). Interfaces have an independent status. Interfaces can be associated with system components in various ways. It is often the case that a component 'has' one or more interfaces. The precise technical meaning of 'has' may vary. We will always assume that for something to be called a component, it must at least be clear which interfaces it has.

Familiarity with the range of services offered by the interface of a system component enables another component to interact with it via the instructions in the interface.

[4] We use 'program component' independently of the phrase 'software component'. There is no implication that a program component contains a program in binary form.

3.1.1 Interface polarization

Program algebra, coprogram calculus, projection semantics and related topics are based upon polarized interfaces. The polarity of an interface determines whether it contains requests or services.

Interface polarization refers to the assumption that all interfaces can be split into the following two polarities: service versus co-service. We will assume that all interfaces are polarized until further notice.[5]

3.1.2 Upper interfaces and lower interfaces

An upper interface[6] is an interface containing all actions that can be called by another component. A lower interface[7] of a component is an interface containing all actions for which a request may be issued by another component. Co-operation between two system components will, as a rule[8], involve an action in the lower interface of an initiating component[9] and in the upper interface of a called component.[10]

3.1.3 Service interfaces and co-service interfaces

Service versus co-cervice expresses a complementarity in the following sense: given an interface that offers instructions as an external service, the complementary co-interface is an interface along which another system component may call these services. Co-operation between components requires an appropriate match between service interfaces and co-service interfaces. Service interfaces are included in the upper interface of a component. Co-service interfaces are included in the lower interface of a component.

3.2 Interface roles

Both service interfaces and co-service interfaces can be connected with different purposes. We will present some important roles. Being a target interface is a possible role of a co-service interface. Similary being a co-target interface is a possible role of a service interface.

[5]Non-polarized interfaces are the point of departure for process algebra and trace theory. Non-polarized interfaces are instrumental for the description of parallel systems.

[6]The upper interface of a component equals the union of its service interfaces. In other words: the upper interface of a component is its maximal service interface.

[7]The lower interface of a component equals the union of its co-service interfaces. In other words: the lower interface of a component is its largest co-service interface,

[8]This asymmetry can be dropped if interface polarization is not assumed.

[9]Note: program components have no upper interface and coprogram components have no lower interface.

[10]If the upper interface of a component is empty, the component is said to lack an upper interface. The symmetric convention holds for lower interfaces.

3.2.1 Target interfaces and co-target interfaces

A target interface of a system component is a co-service interface that is used to issue requests for external action. The processing of requests from a target interface will lead to the production of a boolean reply value, the main reason for the request being connected with other expected effects.

A co-target interface[11] is a service interface through which another component can enact services with external effects.

3.2.2 Reply service interfaces and reply co-service interfaces

A reply service interface is a service interface containing instructions that exclusively invoke state transitions within the component to prepare for the adequate delivery of boolean replies. Coprogram components have a reply service interface only.

A reply co-service interface is a co-service interface containing actions whose only role it is to ask for a reply or to update the state of the recipient of the instruction as a preparation for future actions from the reply co-service interface.

3.3 Interface notation

For a system component C, the union of all of its service interfaces is denoted by $\Sigma_u(C)$, and the union of all its coservice interfaces is written: $\Sigma_l(C)$. If there is at most one service interface, $\Sigma_u(C)$ is also called the service interface of the component C, and if there is at most one lower interface, $\Sigma_l(C)$ is also called the co-service interface of C.

3.3.1 Pseudo co-service interfaces

Programs as such do not have an interface, the difficulty being that it is not clear which superset of the collection of actions featuring in a program behavior should be considered the correct interface. The minimal collection Σ of atomic actions containing all actions that are present in the text of X[12] is denoted with $\Sigma_l^{ps}(X)$, the collection being called a pseudo interface.

To see why $\Sigma_l^{ps}(X)$ is a pseudo interface rather than a proper one consider this example:

$$X = a(124); +b(358); \#\#67; -a(98); a(358).$$

Now $\Sigma_l^{ps}(X) = \{a(98), a(124), a(358), b(358)\}$ which is very implausible as an interface. A reasonable interface might be: $\{a(n), b(n) \mid 0 < n < 1000\}$. This leads to the introduction of the program component $< \{a(n) \mid n < 1000\}, X >$.

[11] A co-target interface is also called API, for Application Programmer Interface.

[12] A reference to program objects is needed if all formal defintions are to be presented.

3.3.2 Behavior pseudo co-service interfaces

Just as it is reasonable to associate a pseudo co-service interface with a program, it is meaningful to associate a pseudo co-service interface with a program behavior. It simply contains all actions that the behavior performs. We omit the precise definions. For a behavior P we denote its pseudo co-interface with $\Sigma_l^{ps}(P)$. If X is a PGLZ[13] program with behavior $|X|_{pglz}$ then $\Sigma_l^{ps}(|X|_{pglz}) \subseteq \Sigma_l^{ps}(X)$. A strict inclusion may hold, e.g. consider the PGLA program $X = !; a$.[14]

3.4 Comment: stratified objects

Of course, unlike a program component or a coprogram component, a system component can have both non-empty upper and lower interfaces. If the two are disjoint the component is called a stratified object; if there is an overlap between the two interfaces, the component is called a recursive (or non-stratified) object.

In particular, if $\Sigma_u(C) \cap \Sigma_l(C) \neq \emptyset$, it makes sense to assume that instructions in $\Sigma_u(C) \cap \Sigma_l(C)$ are to be processed by the component proper, that interpretation constituting a form of recursion.

4 Coprograms and coprogram calculus

In this section a small collection of important coprograms is produced as well as the operator set of coprogram calculus.

4.1 Four abstract data type coprograms

The examples below are considered fundamental because they are useful in a number of circumstances. The examples and their names[15] may be considered the initial segment of a library of useful coprograms. The specifications are informal, but can of course be formalized in much more detail.

Boolean register: `cpbr`. The service interface of `cpbr` consists of:
`{cpbr.setc(true), cpbr.setc(false),`
`cpbr.eqc(true), cpbr.eqc(false)}`.
The reply function will reply `false` to the test instructions
`cpbr.eqc(true)` and `cpbr.eqc(false)`
until the first `br.setc(true)` instruction is called. Thereafter the state of the register starts initialized with `true`. At each request of the form `br.setc(true)` or `br.setc(false)` the reply function replies `true`.[16] The state of the coprogram is flipped whenever it receives a set instruction to the opposite truth value. The test instructions do not change the state

[13] PGLZ denotes an 'arbitrary' program notation, under the restriction that a behavioral semantics in the style of program algebra does exist.

[14] The interface condition for program components can be stated as follows: for $< \Sigma, X >$ to be a component (with X a PGLZ program) it is required that $\Sigma_l^{ps}(|X|_{pglz}) \subseteq \Sigma$.

[15] All (and only) coprogram names starting with `cp` have a public status.

[16] This is an arbitrary default reply for an action that only serves to update the register.

but will return a reply `true` whenever the test instruction mentions the current value of the register.

NN register: `cpnr`. The NN register can contain arbitrary natural numbers: its service interface is:{`cpnr.setc(i)`, `cpnr.eqc(i)` | `i` ∈ NN}. The reply function is self-evident. It generalizes the reply function of `cpbr` to NN instead of {`true`,`false`}.

NN counter: `cpnc` The NN counter `cpnc` has the following service interface: {`cpnc.succ()`,`cpnc.pred()`,`cpnc.isZero()`}. Again the reply function is clear from the mnemonics of the interface actions, as is the convention to reply `true` as a default, on the assumption that the counter is initialized in the empty condition. In particular it is understood that, having become zero, the action `cpnc.pred()` leaves its argument at zero.

NN stack: `cpns` The NN stack `cpns` has the following instruction set as its service interface: {`cpns.pushc(i)`,`cpns.pop()`,`cpns.topEqc(i)` | `i` ∈ NN}. The reply function is clear from the mnemonics of the interface actions, as is the convention to reply `true` as a default, under the assumption that the stack is initialized in the empty condition.

4.2 Coprogram calculus

Five operators and a constant for coprograms are useful in many cases: service interface (Σ_s), non-interfering combination ($H_1 \oplus H_2$), name refinement (`f`.H), restriction ($\Sigma \triangle H$), export ($\Sigma \square H$), which is complementary to restriction, and the empty coprogram ($\emptyset$). The discussion of restriction and export is postponed to the following section.

4.2.1 Service interface operator

If the coprogram H consists of the pair $< \Sigma, F >$ then $\Sigma_s(H) = \Sigma$. The service interface of a coprogram is an upper interface, because the coprogram will not take the initiative in carrying out an instruction from the interface, but will leave that to other parties. Furthermore, it is a reply service interface, the calling party only having an interest in the returned boolean values.

4.2.2 Name refinement

Name refinement (`f`.H) prefixes all names of a coprogram (H) with `f`. for some name `f`. As an example consider the boolean register (`br`) from paragraph 4.1. If several copies of this coprogram are needed one may use:

`f1.cpbr,f2.cpbr,f3.cpbr.`

The interface actions of `f1.cpbr` are

{`f1.cpbr.setc(true)`,`f1.cpbr.setc(false)`,...}.

4.2.3 Non-interfering combination

The non-interfering combination operator takes two coprogram arguments, say H_1 and H_2. The upper interface consists of those actions occurring in only one of the two service interfaces:

$$\Sigma_u(H_1 \oplus H_2) = (\Sigma_u(H_1) \cup \Sigma_u(H_2)) - (\Sigma_u(H_1) \cap \Sigma_u(H_2)).$$

Let F_i be the reply function for H_i. The reply function F for $H_1 \oplus H_2$ inspects the last instruction of its argument list. If that instruction is from H_i all instructions of H_{3-i} (i.e. in $\Sigma_u(H_{3-i})$ are removed and the reply is computed by means of an application of F_i to the remaining list.

4.2.4 Empty coprogram

The empty coprogram is denoted with $\emptyset$. This is the unique coprogram H with $\Sigma_u(X) = \emptyset$, playing but a formal role in the calculus of coprograms.

4.2.5 Calculation rules

Several identities concerning non-interfering combination are valid:

$$H \oplus \emptyset = H, H \oplus H = \emptyset$$

$$H_1 \oplus H_2 = H_2 \oplus H_1$$

$$\mathtt{f}.(H_1 \oplus H_2) = (\mathtt{f}.H_1) \oplus (\mathtt{f}.H_2)$$

$$(H_1 \oplus H_2) \oplus H_3 = H_1 \oplus (H_2 \oplus H_3) \text{ if } \Sigma_u(H_1) \cap \Sigma_u(H_2) \cap \Sigma_u(H_3) = \emptyset$$

4.2.6 Comment

The non-interfering combination provides a disjoint combination of the facilities of the two coprograms, provided their service interfaces are disjoint. If the service interfaces overlap an ambiguity must be avoided and for that reason actions in the overlap are not offered by the non-interfering combination. The facilities of several copies of the same coprogram can be combined after preparatory name refinement, for instance:

$$\mathtt{f1.cpbr} \oplus \mathtt{f2.cpbr} \oplus \mathtt{f3.cpbr}.$$

4.3 Restriction and export for coprogram calculus

The restriction $(\Sigma \triangle H)$ of coprogram H is obtained by removing all interface actions in Σ from the interface of H and by dropping all input streams featuring an action in Σ. The export operator does the converse: it drops all interface actions outside Σ. For restriction and export there are several universally valid identities.

4.3.1 Distribution identities

$$\Sigma \triangle (H_1 \oplus H_2) = (\Sigma \triangle H_2) \oplus (\Sigma \triangle H_1)$$

$$\Sigma \square (H_1 \oplus H_2) = (\Sigma \square H_2) \oplus (\Sigma \square H_1)$$

4.3.2 Special cases

$$\Sigma_u(H) \square H = H, \quad \emptyset \triangle H = H, \quad \emptyset \square H = \emptyset$$

$$\Sigma_u(H_1) \square (H_1 \oplus H_2) = H_1 \ \texttt{if} \ \Sigma_u(H_1) \cap \Sigma_u(H_2) = \emptyset$$

4.3.3 Interaction with the service interface operator

$$\Sigma_u(\Sigma \triangle H) = \Sigma_u(H) - \Sigma, \quad \Sigma_u(\Sigma \square H) = \Sigma \cap \Sigma_u(H)$$

$$\Sigma \triangle H = (\Sigma_u(H) - \Sigma) \square H$$

$$\Sigma \square H = (\Sigma_u(H) - \Sigma) \triangle H$$

5 Interfacing programs and coprograms

Program components have a co-service interface and coprograms have an service interface. Whenever there is a non-empty overlap between the co-service interface of a program component and the service interface of a coprogram, the two components can be interfaced.

The subject of interfacing being open-ended in nature, our discussion focusses on two especially important cases. Let X^i be a program component (with program X and lower interface $\Sigma_l(X^i)$) and let H be some coprogram.

- Use. If $\Sigma_l(X^i) \not\subseteq \Sigma_u(H)$, we may regard $\Sigma_l(X^i) - \Sigma_u(H)$ as a target interface for X^i, $\Sigma_l(X^i) \cap \Sigma_u(H)$ as a reply co-service interface for X^i and as also a reply service interface for H.

  The task of H is to support X^i in its operation, which will express its merit by calling actions through the target interface $\Sigma_l(X^i) - \Sigma_u(H)$. Upon termination of the execution of X, H is forgotten and so is the state it is in.

- Apply. If $\Sigma_l(X^i) \subseteq \Sigma_u(H)$, we may regard $\Sigma_l(X^i)$ as a target interface for X^i. At the same time $\Sigma_l(X^i)$ plays the role of a co-target interface of H.

  Upon termination of the execution of X, the result of the computation materializes in the state of H. The task of X is to transform the state of H.

Both cases 'use' and 'apply' will be captured by means of a special purpose operator. In the case of 'use', the special operator produces the behavior of the program alongside its co-target interface. In the case of 'apply', the special operator determines the state in which the program leaves the coprogram after termination.

5.1 The use-operator and the apply-operator

The use-operator combines a program behavior P and a coprogram H producing a behavior, written P/H. Use refers to the fact that the program issues instructions to the coprogram only for the sake of getting replies returned.

The apply-operator connects a program behavior and a coprogram in an alternative fashion (and under different conditions). It requires $\Sigma_l^{ps}(P) \subseteq \Sigma_l(H)$. The apply-operator is written $P \bullet H$, for a behavior P and a coprogram H. The apply-operator yields a coprogram (or in case of divergence or error D and M respectively).

Both the use-operator and the apply-operator are extended to take progams rather than behaviors as their first arguments. We will only use that extension in combination with the behavioral abstraction operator $|-|$. A subscript will then indicate the program notation of the program and the notation works as follows:

$$|X/H|_L = |X|_L/H$$

$$X \bullet_L H = |X|_L \bullet H.$$

The difference between the use-operator and the apply-operator is a matter of perspective: the use-operator considers the coprogram a transformer of behaviors (or programs) and determines another behavior, whereas the apply-operator considers the program a transformer of coprograms and therefore produces a coprogram (unless some error occurs). In the perspective of the apply-operator the coprogram is input and the modifications that are applied to it during a computation are the essence. The use-operator simply drops the coprogram after program termination. Use-operators are meaningful only for programs using basic actions that are not contained in the co-service interface of the component.

5.2 Repeated use and apply

It is reasonable to repeatedly use the use-operator. If $\Sigma_u(H_1) \cap \Sigma_u(H_2) = \emptyset$, one may write:

$$(X/H_1)/H_2 = X/(H_1 \oplus H_2).$$

Similarly a repreated application of the apply-operator is possible. Under strict conditions demanding that ';' represents sequential composition between X_1 and X_2, the following is valid:

$$X_1 \bullet (X_2 \bullet H) = (X_2; X_1) \bullet H.$$

5.3 Semantic equations for use and apply

Behaviors suitable for the semantics of program algebra expressions have been discussed in [4]. A summary is given here.

5.3.1 Program behaviors

M denotes the behavior shown by a program in error. This will be caused by the evaluation of an abort instruction. D is the behavior of a non-terminating or divergent program. A diverging program may not even perform tests while diverging. The most typical source of divergence is a cyclic succession of jump or goto instructions. $||!||$ is the irreducible behavior of the successful termination instruction, $||a||$ denotes the primitive behavior describing the execution of a single atomic action, and $Y(a)$ is a predicate (or condition) predicting whether the boolean-returning action a will return `true` or `false`. Further $P \cdot Q$ represents the sequential composition of behaviors P and Q. Conditions are used to combine behaviors as follows: if P and Q are behaviors, and if a is a boolean-returning atomic instruction, $P \lhd Y(a) \rhd Q$ behaves like P if $Y(a)$ holds true and like Q if it does not. Program behaviors will always end (if at all) with M, D or $||!||$.

5.3.2 Coprogram effect notation

Let P be a program behavior and let H be a coprogram. We wish to define P/H as the behavior of P using H. The actions in the lower interface of P that are also in the upper interface of H will be processed by H. An auxiliary notation has to be introduced beforehand. For a coprogram $H =< \Sigma, F >$ and an action a in its upper interface the coprogram $H_{\partial a}$ is defined by

$$H_{\partial a} =< \Sigma, F' >$$

with $F'(w) = F(a, w)$ for all interface action sequences w.

5.3.3 Semantic equations for the use-operator

The defining rules for P/H are these:

$$||!||\,/H = ||!||$$

$$M\,/H = M,\ D\,/H = D$$

$$(||a|| \cdot P)/H = ||a|| \cdot (P/H) \texttt{ if } a \notin \Sigma_u(H)$$

$$(||a|| \cdot P)/H = (P/H_{\partial a}) \texttt{ if } a \in \Sigma_u(H)$$

$$(P \lhd Y(a)) \rhd Q)/H = (P/H) \lhd Y(a) \rhd (Q/H) \texttt{ if } a \notin \Sigma_u(H)$$

$$(P \lhd Y(a) \rhd Q)/H = (P/H) \lhd H(a) \rhd (Q/H) \texttt{ if } a \in \Sigma_u(H).$$

These equations determine a behavior by means of a step-wise processing of the actions of the program. Only the actions and tests outside $\Sigma_u(H)$ will contribute

to the resulting behavior. If such an action cannot be found, this may be due to non-termination. Non-termination is then expressed by means of the equation: $P/H = D$.

5.3.4 Equations for the apply-operator

D and M are considered coprograms as well if the reply function yields D or M represented by the coprogram being identified with D or M respectively.

$$||!|| \bullet H = H$$

$$M \bullet H = M,\ D \bullet H = D$$

$$(||a|| \cdot P) \bullet D = D,\ (||a|| \cdot P) \bullet M = M$$

$$(||a|| \cdot P) \bullet H = (P/H_{\partial a}) \text{ provided } H \neq D \text{ and } H \neq M$$

$$(P \lhd Y(a) \rhd Q) \bullet H = (P \bullet H) \lhd H(a) \rhd (Q \bullet H)$$

If application of these rules fails to lead to a converging computation, the computation of P on H is said to diverge, which is written as $P \bullet H = D$.

5.4 How use is used and how apply is applied

The use-operator can play an key role in projection semantics. In some cases a projection for a program notation can only be found at the cost of the introduction of an auxiliary coprogram which the projected program may use.

The apply-operator plays a key role in the description of batch processing. In the formalization of batch processing both inputs and outputs of programs are packed in a coprogram. A batch program is seen as a coprogram transformer, semantically captured by the apply-operator.

6 Concluding remarks

The symmetry and complementarity of programs and coprograms is based upon the adoption of polarized interfaces. Polarized interfaces are to program algebra what arbitrary interleaving is to process algebra, a hypothesis which is speculative and productive at the same time.

Just as ACP-style process algebra can be understood as a project designed to exploit the arbitrary interleaving hypothesis, program algebra may be regarded as a theory committed to the exploitation of the conceptual limits of the interface polarization hypothesis.

If the interface polarization hypothesis is refuted, programs and coprograms can be regarded as instances of the same category: processes.

References

[1] J.A. Bergstra. Datatypen bezien vanuit de recursietheorie. In J.C. van Vliet, editor, *Colloquium Capita Datastructuren*, pages 157–170. Mathematisch Centrum, Amsterdam, 1978.

[2] J.A. Bergstra. What is an abstract data type? *Information Processing Letters*, 7(1):42–43, 1978.

[3] J.A. Bergstra and J.-J.Ch.Meyer. Equational specification of finite minimal unoids, using unary hidden functions only. *Fundamenta Informaticae*, 5(2):143–170, 1982.

[4] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998.

[5] L.M.G. Feys, H.B.M Jonkers, and C.A. Middelburg. *Notations for Software Design.* Springer Verlag, 1994.

A Specification techniques for coprograms

Coprograms primarily being behaviors, the description of a coprogram in general poses two questions at the same time: (1) which specification technique should be used, (2) how should the technique be used in a particular case.

A.1 Construction versus specification

The simplest way of denoting a program is to write it. Programs are typically constructed by means of the successive application of a limited number of construction principles to a limited number of primitives.

Some other mathematical objects, for instance the structure of the real numbers in classical mathematics, are specified rather than constructed. Specification is performed by means of the presentation of a list of properties.

The most obvious category of objects admitting description by construction are the natural numbers. Numbers also allow specification : the construction $n = 97$, corresponds to 'n equals the largest prime number below 100'. The second description is typically a specification.

A.2 Specification styles

As it turns out coprograms lack any direct means of construction. Coprograms must be specified. We list some options.

Informal specification. An informal specification of a coprogram (or of a class of coprograms) will be definite about the interface of instructions that

can be issued to the coprogram.[17] The informal part is in the description of the reply function. There is no doubt that the combination of a precise interface description with an intuitively appealing description of a reply function is very useful in practice.

Property-oriented specification. This technique includes many more specialized techniques using restricted logics. Point of departure is the observation that a coprogram is itself a three-sorted algebra: a sort (Σ) of interface actions with a constant for each interface action, a sort (B) of booleans, equipped with constants for both truth values, and a sort (Σ^+) consisting of non-empty sequences of interface actions, the reply function being a function in this algebra.

Property-oriented specification comprises writing an axiomatization for this three-sorted structure, preferably disallowing auxiliary operators.

Property-oriented coprogram specifications in first-order logic are difficult to find, even for remarkably simple coprograms. If auxiliary interface actions are allowed it is always possible to find a direct specification of a coprogram using equations only.[18] However, the proof of this fact is long and complicated and the practical implication of the result is not immediately clear. (We refer to [3] for the proof.)

Temporal logics can be used to provide property-oriented coprogram specifications, in some cases with remarkable elegance.

State-space model description. A state-space model for a coprogram allows adding another auxiliary sort (S) to the sorts mentioned above. This sort contains states in which the coprogram may be during the execution of a succession of interface actions. For the sort of states a constant and three operations are needed:

1. the initial state constant $\mathtt{s}_{init}$ determines in which state the coprogram will start its operation upon initialization,
2. the effect function $\mathtt{E} : S \times \Sigma \to S$ takes a state and an interface action and determines the state in which the coprogram will be after processing the interface action,
3. the boolean result function $\mathtt{Y} : \Sigma \times S \to \{\mathtt{true}, \mathtt{false}, M, D\}$ determines which reply is produced when state s is reached after processing an interface action,
4. the cumulative effect function CE determines the state of the coprogram after a sequence of interface actions has been processed.

[17] With CORBA IDL industry has produced a description technique for such interfaces that may well be sufficiently strong for many years to come.

[18] It must be assumed that the coprogram is computable. The case of a finite state-space is especially intricate.

The connection between these operators and the coprogram $H =< \Sigma, F >$ being specified is then as follows:

$$\mathtt{CE}([\,]) = s_{init}$$

$$\mathtt{CE}(w, a) = \mathtt{E}(\mathtt{CE}(w), a)$$

$$F(w, a) = \mathtt{Y}(a, \mathtt{CE}(w))$$

Obviously the operators CE and F can be derived when E and R are known. Therefore these techniques usually make no mention of any operators other than E and R.

Technically a state-space description of a coprogram amounts to a specification of the algebra with sorts Σ, S and B and operations E,CE,R. It turns out that if additional functions are allowed, the use of equations will suffice to obtain appropriate specifications of the state-space model in all relevant cases.

From a certain level of complexity, state-space models are the only option if a formal description of a coprogram is needed. Regarding state space model descriptions the following remarks can be made.

A.2.1 Parametrized coprograms

The notation for state space model descriptions of coprograms can be made more explicit, thus obtaining a parametrized family of coprograms including the coprogram H. Using the notation given above each state $s \in S$ determines a coprogram $H_s =< \Sigma, F_s >$. The definition of F_s makes use of an auxiliary operation $\mathtt{CE}_s$:

$$\mathtt{CE}_s([\,]) = s$$

$$\mathtt{CE}_s(w, a) = \mathtt{E}(\mathtt{CE}_s(w), a)$$

$$F_s(w, a) = \mathtt{Y}(a, \mathtt{CE}_s(w))$$

The connection between H and this coprogram family reads $H = H_{s_{init}}$.

A.2.2 Notational conventions for effect functions

In practical use the format given here is often present in a disguised form only. Rather than having an effect function E with a second argument in Σ, a special operation (say e_a) will be used for each $a \in \Sigma$. If a has parameters these will be taken as additional parameters for e_a, usually preceding the state parameter. Of course a more mnemonic notation than e_a is likely to be used. The yield operation is usually only specified for non-void actions, void actions being ones that have yield `true` by definition. In most cases non-void actions cannot change the state. Then it suffices to describe these by means of a mapping y_a computing a boolean value from a state argument. Again a more mnemonic notation is likely to be used, and y_a may be given the parameters of a as additional parameters.

A.3 Non-monotonic rewriting, a satisfactory solution

The proof that equations are sufficient as a specification technique, is much more natural for state-space models of computable coprograms than for property-oriented specifications. In the latter, auxiliary functions will be needed. Nevertheless, it seems that in practice the use of a purely equational format is insufficient. Several additional features such as conditional equations, matching conditions, equations with priorities and default equations have been proposed and have proven extremely useful in this context. Together these features constitute non-monotonic rewriting, an interesting mix of logic programming and functional programming. Non-monotonic rewriting appears to be quite satisfactory as a specification method for complex algebras. Much more powerful, but less amenable to automatic prototyping, are the specification techniques discussed in [5]

A.3.1 Subject oriented programming

Programming by means of the construction of modular non-monotonic term-rewriting systems may be called subject-oriented programming. Theory extension is the main tool of subject-oriented programming. By nature subject-oriented programming is far more abstract than object-oriented programming. Object-oriented programming is often committed to the perspective of an object being a distinguished (and somehow logically coherent) subspace of the machine memory during program execution. Class extension extends a predefined bookkeeping mechanism from one class to another one. In subject-oriented programming programmer defined theories are extended. Needless to say the implementation problems for subject-oriented programming are formidable.

Projection semantics for recursion and for multi-threading

J.A. Bergstra* & M.E. Loots†

March 30, 1999

Abstract

Primitives are given for the introduction of recursion and multi-threading in the context of the program notation PGLDg. For both extensions a projection semantics is given. Both projections require an auxiliary coprogram.

1 Introduction

For readers of this paper familiarity is assumed with the concept of projection semantics as it has been outlined in [1], the program notation PGLDg from [2], the concept of coprogram, the particular coprogram cpns, encoding a stack of natural numbers, and the use-operator for interfacing programs and coprograms (see [3]). The present objective is to design primitives for recursion and for multi-threading on the basis of PGLDg and to provide a projection semantics for both extensions of PGLDg.

The authors acknowledge a number of useful comments made by Alban Ponse on the basis of a draft of this paper.

2 Extensions of PGLDg

PGLDgr is an extension of PGLDg with two primitives capturing the essence of recursion, PGLDgmt being an extension of PGLDg capturing key aspects of multi-threading and PGLDgμ being an extension of PGLDg capturing the mechanism of dynamic goto's. PGLDgμ will be needed as an intermediate stage for the projection semantics of PGLDgr and of PGLDgmt. It being the simplest of the three by far, its details are presented first. Subsequently PGLDgr and PGLDgmt are introduced.

*University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl

†Tilburg University, Faculty of Philosophy, email: M.E.Loots@kub.nl.

2.1 Advanced control instructions

Some general properties of PGLDg extensions should be discussed in order to motivate certain design decisions for the new primitives.

2.1.1 PGLY_Σ, instruction sequences with basic actions

PGLY will stand for any extension of PGLDg. Its programs consist of finite sequences of instructions. These instructions can be a, $+a$, $-a$, with a taken from the basic action parameter Σ of the language. However, they can also be PGLDg control instructions (in particular: $\#$, $\pounds k$, $\#\#\pounds k$), instructions specific for PGLY, or instructions in specified coprogram service interfaces. The instructions specific for PGLY are called advanced control instructions, control because they are not basic actions or tests, advanced because they are not in PGLDg. For each parameters set Σ there is an instance PGLY_Σ of the language PGLY. Alle extensions of PGLDg involve additional instructions. Each instruction which is not in the parameter set Σ of atomic actions and not in the syntax of PGLDg is called an advanced control instruction. We will assume that basic actions from Σ are always different from the advanced control instructions.

2.2 Coprogram control instructions

Often it is important to permit a language to make use of the actions in the service interface $\Sigma_l(H)$ of some coprogram H. Although such actions are best regarded as part of the parameter set Σ, bookkeeping concerns suggest having them outside Σ in their own class of basic actions: coprogram control actions. We will use some coprograms below (the stack and the thread vector), collecting all of their interface actions in $\Sigma_{l_{max}}$. Parameter sets Σ are kept disjoint from $\Sigma_{l_{max}}$.

2.3 Coprogram based language extensions

For a program notation PGLY and a coprogram CPZ, the program notation PGLY/CPZ allows PGLY programs to contain instructions from the service interface of CPZ (denoted with cpz inside formulas.) Under the assumption that coprogram service interface instructions are outside Σ, $\mathrm{PGLY/CPZ}_\Sigma$ denotes all programs of this extension of PGLY containing basic instructions from Σ. The behavior of PGLY/CPZ program X is derived from its behavior as a PGLY program as follows:

$$|X|_{pgly/cpz} = (|X|_{pgly})/\mathtt{cpz}$$

Here P/H is the use-operator of [3], instantiated with $P = |X_{pgly}|$ and $H = \mathtt{cpz}$. It follows that if the behavior of PGLY programs is known, then the behavior of PGLY/CPZ programs is known as well. We will make use of this observation for PGLX = PGLDg and for two manifestations of CPZ; CPZ = cpns, the coprogram containing a natural number stack, and CPZ = cptc, a coprogram

capturing the thread vector emerging from the description of a multi-threading extension of PGLDg.

2.4 Generic formulation of the objective of projection semantics

The object of this paper is defined by the following problem statement. Let PGLY be any of the languages PGLDgr or PGLGgmt. Assume that a reasonably clear intuitive account for the meaning of PGLY programs has been provided along with the description of its syntax (in particular its advanced control instructions). Formalize the semantics of PGLY by means of the construction of:

- a coprogram `cpz`, and
- a transformation `pgly2pgla/cpz`, (called a projection), transforming arbitrary PGLY_Σ programs to PGLA_Σ programs making use of actions from the interface of `cpz`.

in such a way that the following identity can be considered an adequate formal definition of the behavior of a PGLY program X:

$$|X|_{pgly} = |\texttt{pgly2pgla/cpz}(X)|_{pgla/cpz}$$

2.4.1 Using intermediate program notations

A projection `pgldg2pgla` from PGLDg to PGLA being known, it suffices to find a projection `pgly2pgldg/cpz` from PGLY to PGLDg such that

$$|X|_{pgly} = |\texttt{pgly2pgldg/cpz}(X)|_{pgldg/cpz}$$

We will make use of an extension PGLDgμ of PGLDg and its projection `pgldgmu2pgldg`. Given this projection, the task of finding a projection `pgly2pgldg/cpz` is reduced to the task of finding a projection `pgly2pgldgmu/cpz` such that

$$|X|_{pgly} = |\texttt{pgly2pgldgmu/cpz}(X)|_{pgldgmu/cpz}$$

The projection `pgldgmu2pgldg` provides a semantics for `pgldgmu/cpz` programs Y in the following way:

$$|Y|_{pgldgmu/cpz} = (|\texttt{pgldgmu2pgldg}(Y)|_{pgldg})/\texttt{cpz}$$

Of course, the more expressive language one uses as a target of the projection, the easier it becomes to define it.

2.5 Rationale of projection semantics

All descriptions of a programming notation PGLY must provide some form of behavioral semantics for PGLY programs. This is necessary in case reliable sofware is to be produced by means of writing PGLDg programs.

The precise concept of behavior left aside, the difficulties involved in finding an adequate definition of behavior are often almost unsurmountable.

Many techniques have been designed to give definitions for language semantics. It is often considered vital to base such a technique on the right kind of semantic concept (i.e. the right, or best concept of behavior). Unfortunately, that viewpoint makes semantic descriptions dependent on a formidable parameter: the mathematical meta-theory of the semantic concepts used. The rationale for the use of projection semantics can be summarized in the following points:

1. Obtain maximal independence from the conceptual meta-theory regarding behaviors. The behavioral semantics of PGLA is a 'back end' of projection semantics with considerable flexibility.

2. Make maximal use of what is known about compilers for a language (PGLY).

3. Use a modular and bottom-up style of development, allowing a maximum re-usage of projections already known.

4. Take care that projection semantics descriptions be executable (i.e. automatic generation of a prototype compiler is possible).

5. Remove any incentive (or at least any need) to 'discover' semantic concepts the designers of a language did not have in mind. The construction of suitable 'domains' should be restricted to cases that are really worthwhile.

2.6 A design rule

The general design rule is to avoid test actions being immediately followed by advanced control instructions. The rationale for this restriction, imposed on all language extensions PGLX of PGLDg, is to avoid confusion when a transformation expands the advanced control instruction to a sequence of instructions, the confusion being about where to resume computation after a negative outcome of the test.

3 Three extensions of PGLDg

We will now discuss the syntax of three extensions of PGLDg, beginning with the extension with the goto/searh instruction. This advanced control instruction initiates a search, leading to the identification of a natural number which then can be used as the target label of a goto instruction.

3.1 A notation with dynamic goto's: PGLDgμ

For technical reasons a simple extension of PGLDg is very useful. This extension introduces a single new advanced control instruction, under the assuption that a certain family of instructions may be used. The construction is called 'goto/search instruction', appearing in two forms: positive and negative. [1] Let $\varphi(j)$ be a test instruction dependent on a natural number j. Then $+\varphi(0), -\varphi(0), +\varphi(1), -\varphi(1), +\varphi(2), -\varphi(2), \ldots$ are test instructions. The positive and negative goto/search instructions for $\varphi(j)$ are:

- $\#\#\pounds[\mu\, j. + \varphi(j)]$
- $\#\#\pounds[\mu\, j. - \varphi(j)]$

The language PGLDgμ permits the use of positive and negative goto/search instructions, in addition to the instructions of PGLDg. These are considered advanced control instructions and should not be used immediately following a test instruction.

3.1.1 Projection semantics for PGLDgμ

The intermediate language PGLDgμ needs a projection semantics of course. The translation of programs works statement by statement. Only the (new) advanced control instructions are modified: in particular, if k is the numerical value of the largest label occurring in a program to be translated, the transformation of the positive goto/search instruction is as follows: `pgldgmu2pgldg`$(\#\#\pounds[\mu\, j. + \varphi(j)]) =$

$$+\varphi(0); \#\#\pounds 0; +\varphi(1); \#\#\pounds 1; \cdots; +\varphi(k); \#\#\pounds k.$$

In words: decide whether $\varphi(0)$ holds; if so, goto $\pounds 0$, if not, consider $\varphi(1)$, etc. For the negative case we find: `pgldgmu2pgldg`$(\#\#\pounds[\mu\, j. - \varphi(j)]) =$

$$-\varphi(0); \#\#\pounds 0; -\varphi(1); \#\#\pounds 1; \cdots; -\varphi(k); \#\#\pounds k.$$

The projection `pgldgmu2pgldg` translates goto/search instructions in a context-depending way. The more label catches a program contains, the more tests the translated goto/search instruction will have to perform. If needed, different translations of goto search can be mapped onto a single occurrence of the sequence, thus ensuring better code compactness after projection.[2]

The general nature of the projection `pgldgmu2pgldg` implies that it will work in the presence of arbitrary coprograms. It induces projections

`pgldgmu/cpns2pgldg/cpns`, and `pgldgmu/cptv2pgldg/cptv`.

[1] The goto/search instruction is an implementation of a dynamic jump in the context of goto's and labels.

[2] For this reason the trailing # has been added. Failing that abort instruction, code sharing cannot always be achieved. If processing speed matters and additional coprograms can be used, a binary search can replace the linear search, thus making the program larger in size but faster at work.

3.2 Primitives for recursion: PGLDgr

The introduction of recursion on top of PGLDg can be achieved by introducing two new instructions: the returning goto instruction and the return instruction. The syntax proposed for these instructions is as follows:

- $\rho\#\#\pounds k$
- ρ

The returning goto instruction '$\rho\#\#\pounds k$' implies a goto together with the command to jump to the instruction immediately following itself whenever a return instruction is reached in the computation from $\pounds k$. The return instruction 'ρ' must be executed by jumping back to the the control point just after the last returning goto instruction to which a return has not yet taken place.

Of course, PGLDgr allows the instructions of PGLDg (including some actions from a collection A of atomic instructions) and the two instructions '$\rho\#\#\pounds k$' and 'ρ'. The latter instructions are considered advanced control instructions.

A better explanation can be found by using a stack. There is a stack of control points (labels or instruction numbers). A returning goto instruction first puts the control point just following it on the stack and then issues a goto. A return instruction performs a look- up on the stack and finds the return label on top of the stack, then it issues a goto towards this return label. After that goto has been performed the stack is popped.

3.3 Primitives for multi-threading

Normal program execution can be seen as the single-thread version of multi-thread execution. A thread represents a control point in a program. A multi-threaded program is working at different control points at the same time.

The 'fork thread at label k' instruction results in the creation of a new thread, which starts its computation by making a goto step to label k. The new thread has the same priority as its parent, i.e the thread that was executing the fork instruction. It will be placed at the head of the thread vector.[3] The 'set priority to k' instruction sets the priority of the thread executing it to the natural number k. The syntax proposed for these actions is as follows:

- $\mathtt{fT}\#\#\pounds k$
- $\mathtt{sP}\ k$

[3]There are several versions of the fork, all having to do with the ramification that emerges from different kinds of jumps. The jump or goto that characterizes the fork instruction determines at which control point the new thread will have to start. Besides goto based jumps there are forward jumps, backward jumps and absolute jumps, the ramification of goto's is similar: (i) forward jump fork: $\mathtt{fT}\ \#k$, (ii) backward jump fork: $\mathtt{fT}\ \backslash\#f$, (iii) absolute jump fork: $\mathtt{fT}\ \#\#k$, (iv) forward label jump fork: $\mathtt{fT}\ \#\pounds k$, (v) backward label jump fork: $\mathtt{fT}\ \backslash\#\pounds k$, (vi) absolute label jump fork: $\mathtt{fT}\ \#\#\pounds k$.

3.4 A program notation with multi-threading: PGLDgmt

The program notation PGLDgmt extends PGLDg with the two instructions for multi-threading mentioned above. According to the rules of thumb outlined in [3] these two instructions will be counted as advanced control instructions. Therefore programming in PGLDgmt must be done in such a way that no test is immediately followed by one of these instructions.

4 Projection semantics for recursion primitives

We will translate PGLDgr into PGLDg, with the help of the coprogram NN stack (`nnst`) from [3]. The projection is named `pgldgr2pgldgmu/nnst` and its definition is given by $\texttt{pgldgr2pgldgmu/nnst}(u_1;...;u_n) = \psi_1(u_1);...;\psi_n(u_n)$, where the auxiliary functions ψ_i read as follows:

$$\psi_i(\#\#\pounds l) = \#\#\pounds 2 \cdot l$$

$$\psi_i(\pounds l) = \pounds 2 \cdot l$$

$$\psi_i(\rho\#\#\pounds k) = \texttt{nnst.pushc}(2 \cdot k + 1); \#\#\pounds 2 \cdot k; \pounds 2 \cdot k + 1; \texttt{nnst.pop()}$$

$$\psi_i(\rho) = \#\#\pounds[\mu j. + \texttt{topEqc}(j)]$$

$$\psi_i(u) = u \ \texttt{otherwise}$$

The first and second equation ensure that all labels occurring in the source program are renamed so that no confusion with new labels will arise. This renaming is achieved by doubling the numerical value of the label. Odd labels will be new ones, introduced by the translation. The semantics of a PGLDgr program X is thus determined by:

$$|X|_{pgldgr} = |\texttt{pgldgr2pgldgmu/nnst}(X)|_{pgldgmu/nnst}.$$

5 Projection semantics for multi-threading

A so-called thread vector will contain data concerning the existing threads. Initially there is only a single thread, which is created during the start-up of the program. We will formalize the thread vector as an abstract data type coprogram in the sense of [3]. The formal specification of the thread vector given below is a state-based model specified by means of non-monotonic rewriting. We will focus on the design of the coprogram `cptv` representing the thread vector first.

5.1 The thread vector coprogram

The intuition of multi-threading is very simple. Instead of just a single program counter, a number of program counters points at different control points, each

given the opportunity to progress in turn, turn-taking being scheduled deterministically. A fork instruction allows one to introduce a new thread, termination of a thread leading to a system running with one thread less. Abort of a thread makes the entire program abort. Only if all of its threads have successfully terminated will a multi-threaded program terminate successfully as well.

The coprogram `cptv` that will be used for giving a projection semantics of multi-threading employs a service interface containing 8 (types of) actions. Together with a listing of the interface actions of `cptv` we provide informal explanations of the intended meaning of the interface instructions. The thread vector is a list of thread items. The lists changes in contents and length during a computation. The first item of the list is called the leading item. A thread item consists of a pair of two naturals, the first one representing a priority level, the second one representing a label.

5.1.1 The thread vector service interface

The service interface of `cptv` is as follows:

1. `newItem`(NN)
   adds a new item (l, k), at the second place of the thread vector. Here k is the NN parameter of the action and l is the priority of the current leading item.

2. `delete`()
   removes the leading thread item.

3. `isEmpty`()
   tests whether or not the thread vector is an empty list.

4. `sort()`
   sorts the thread item list on increasing priority of thread items (this instruction is only useful after another `sP`(k) instruction has been performed).

5. `rotate`()
   induces a cyclic shift of all thread items with top priority, where the first element of the sublist of top priority items moves to its tail, the second element in the list moves to the first place and so on.

6. natural `setLdCpl`(NN)
   adapts the current program label value of the leading thread item.

7. `setLdPri`(NN)[4]
   updates the priority of the leading thread item.

8. natural `ldCplEq`(NN)
   decides whether the program label value of the leading thread item equals NN.

[4]NN denotes the set of natural numbers, PNN the set of positive natural numbers.

We will use name refinement and prefix all of these instructions with the name of the coprogram: e.g.`cptv.setLdCpl`(k)

5.1.2 An informal specification of the thread vector

The data structure that is maintained by the virtual machine for multi-threading consists of a thread vector listing a sequence of thread items.

Thread items serve to represent the existence of threads. Thread items are pairs of natural numbers: say (p, l) where p is the current priority level of the thread represented by the item, and l is the current value of the program label of the thread denoted by the item. The program label indicates the label to which the thread will jump if the thread gets its next turn to be active.

There are several primitives for the maintenance of the thread vector. The interface has been mentioned in detail above. Only the test on emptiness and the identity test for the current label of the leading thread produce meaningful (boolean) output. All other actions produce a default boolean output `true`.

Thread vectors can be described by means of an abstract data type specification. The specification below uses three constructors:`item`: NN $\times$ NN $\rightarrow$ TRV, a constant ETV for the empty thread vector, and concatenation NN * NN $\rightarrow$ NN. Concatenation is associative. All operations mentioned in the interface are to be specified on top of these primitives. The rotation instruction deserves some additional comments.

5.1.3 Design options for the thread vector

Whenever a thread has performed an action it is possible to shift the sequence of top priority threads in order to ensure live execution of all threads. This is done by means of the instruction '`rotate()`'. If this maximal priority rotation is not performed it is possible for a thread to starve, in the sense that it never performs an action but is infinitely often (or even permanently at some stage) amongst the threads with top priority. Whether or not rotation is implicit in the execution model of multi-threading is a degree of freedom. In practice it will often be too expensive to perform rotation after every step. It can also be done after every 1000 steps or so. Alternatively rotation can be regarded as a programmable action. Yet another option is to regard the realization of rotations as a responsibility of the underlying machine model. Neither after every step or after a fixed number of steps for a thread, nor at an explicit instruction call from a program can the threads be forced to rotate. The run-time system will ensure a regular interleaving of rotations that are triggered by a so-called system event.

5.1.4 Selection of a formal specification style

We will provide a state-space-oriented formal description, following the conventions given in `BL99c`. The formalization introduces two new sorts: ITM and TRV (for item and thread vector). TRV plays the role of a state space and

ETV is a constant denoting the empty thread vector, serving as the initial state at the same time.

For each instruction in the interface of cptv *there is an effect function in* the formal specification. It has the arguments of the instruction plus an additional argument representing the state. This additional argument is of sort TRV and is listed first. The specification technique used is based on non-monotonic equational term rewriting.

5.1.5 A formal specification of the thread vector coprogram

The specification below uses conditional equations (with positive and negative conditions), default equations and list matching. All equations are of sort TRV, subterms of the form $\mathtt{item}(p, l)$ are of sort ITM. V, V_1, V_2 are variables ranging over the sort TRV. * denotes list concatenation on TRV.

$$\mathrm{ETV} * V = V,\ V * \mathrm{ETV} = V$$

$$\mathtt{newItem}(k, \mathrm{ETV}) = (0, k)$$

$$\mathtt{newItem}(k, \mathtt{item}(p, l) * V) = \mathtt{item}(p, l) * \mathtt{item}(p, k) * V$$

$$(V_1 * V_2) * V_3 = V_1 * (V_2 * V_3)$$

$$\mathtt{delete}(\mathrm{ETV}) = \mathrm{ETV},\ \mathtt{delete}(\mathtt{item}(p, l) * V) = V$$

$$\mathtt{isEmpty}(\mathrm{ETV}) = \mathtt{true},\ \mathtt{isEmpty}(\mathtt{item}(p, l) * V) = \mathtt{false}$$

$$\mathtt{sort}(V_1 * \mathtt{item}(p_1, l_1) * \mathtt{item}(p_2, l_2) * V_2) =$$

$$\mathtt{sort}(V_1 * \mathtt{item}(p_2, l_2) * \mathtt{item}(p_1, l_1) * V_2) \ \ \mathtt{if}\ p_2 < p_1$$

$$\mathtt{sort}(V) = V \ \ \mathtt{otherwise}$$

$$\mathtt{rotate}(\mathtt{item}(p_1, l_0) * V_1 * \mathtt{item}(p_1, l_1) * \mathtt{item}(p_2, l_2) * V_2) =$$

$$V_1 * \mathtt{item}(p_1, l_1) * \mathtt{item}(p_1, l_0) * \mathtt{item}(p_2, l_2) * V_2 \ \ \mathtt{if}\ p_2 \neq p_1$$

$$\mathtt{rotate}(V) = V \ \ \mathtt{otherwise}$$

$$\mathtt{ldCplEq}(\mathrm{ETV}, 0) = \mathtt{true}$$

$$\mathtt{ldCplEq}(\mathtt{item}(p, l) * V, l) = \mathtt{true}$$

$$\mathtt{ldCplEq}(V, l) = \mathtt{false}\ \ \mathtt{otherwise}$$

$$\mathtt{setLdPri}(q, \mathrm{ETV}) = \mathrm{ETV}$$

$$\mathtt{setLdPri}(q, \mathtt{item}(p, l) * V) = \mathtt{item}(q, l) * V$$

$$\mathtt{setLdCpl}(k, \mathrm{ETV}) = \mathrm{ETV}$$

$$\mathtt{setLdCpl}(k, \mathtt{item}(p, l) * V) = \mathtt{item}(p, k) * V$$

5.2 A projection function for PGLDgmt

In this section we will provide a projection from multi-threaded programs to PGLDgμ. Consider PGLDgmt program $X = u_1; \cdots; u_n$; its projection proceeds as follows. We notice that X has instruction length m. Then it is transformed into X' by increasing each of the labels by $m + 2$.[5] This does not affect the be-

[5] G-occurrences as well as L-occurrences should be increased.

havior of the program, but all of its labels are now outside the range $\{0, \cdots, m\}$. This program is subsequently changed so that termination takes place exactly with a jump to label m, (for which there is no corresponding label catch in the transformed program), or by executing the last program instruction. This yields program X''.

Then assume that $X'' = v_1; \cdots; v_m$. On this preprocessed PGLDgmt program a projection into PGLDgμ can be properly defined. The dynamic labels can be projected back to PGLDg, according to 3.1.1. We have used the if-then-else construct with self-evident semantics (however, see [2]). Its projection onto PGLDg was provided in [2].

The projection `pgldgmt2pgldgmu/cptv` is defined on the result X'' of the above preprocessing procedure: `pgldgmt2pgldgmu/cptv`$(X'') =$

$$\texttt{cptv.newItem}(0); \texttt{cptv.setLdCpl}(0);$$

$$\pounds 0; \psi_1(v_1); \pounds 1; \cdots; \pounds m-1; \psi_m(v_m); \pounds m;$$

$$\texttt{cptv.delete()};\texttt{+cptv.isEmpty()}; \#\#\pounds m+1;$$

$$\#\#\pounds[\mu j.\texttt{+cptv.LdCplEq}(j)]; \pounds m+1$$

The auxiliary functions ψ_i for $i \in \{1, \cdots, n\}$ are given below. First we define an abbreviation (H) for a program fragment that will be repeated just before each proper action of a thread is performed. This program tests the thread vector on emptiness; if it is empty, the program must terminate (jump to m) and after rotation it retrieves the label to which a jump must be performed in order to execute the next instruction of the current leading thread, i.e. the thread among those with maximal priority that happens to be at the head of the thread vector.

H $= \texttt{cptv.rotate}(); \#\#\pounds[\mu j.\texttt{+cptv.LdCplEq}(j)]$

The defining equations for the feature projection removing multi-threading follow the case distinction of possible instructions:

void action instruction The action is performed, then the current label of the leading thread is updated. Subsequently a rotate takes place (as the thead vector cannot be empty):
$\psi_i(a) = \texttt{cptv.setLdCpl}(i);$ H

goto instruction. There are two cases: if the jump is into the exit label, a thread terminates and its item must be deleted, otherwise an update of the current leading program label is required.
$\psi_i(\#\#\pounds k) = \texttt{cptv.setLdCpl}(k);$ H

label catch instruction. Immediately after the processing of a label catch instruction the active thread gets another turn:
$\psi_i(\pounds k) = \texttt{cptv.setLdCpl}(i)$

fork instruction. To realise a fork a new item must be made. First, however, the current label is increased by one, because the thread performing the fork will proceed by normal flow of control. Then after the new item has been added, its current label must be set correctly.[6]
$\psi_i(\mathtt{fT}\,\#\#\pounds k) = \mathtt{cptv.setLdCpl}(i);\mathtt{cptv.newItem(k)};\mathrm{H}$

positive test instruction. For the sake of readability the conditional construct is used.
$\psi_i(+a) = \mathtt{if} + a\{;\mathtt{cptv.setLdCpl}(i);\}\{;\mathtt{cptv.setLdCpl}(i+1);\};\mathrm{H}$

negative test instruction. This case is similar to the previous case, mutatis mutandis (i.e. replacing $+a$ by $-a$.)

priority update instruction. The translation simply invokes an update of the leading item priority and advances its program label by 1.
$\psi_i(\mathtt{sP}(k)) = \mathtt{cptv.setLdPri}(k);\mathtt{cptv.setLdCpl}(i);\mathrm{H}$

abort An abort needs no transformation.
$\psi_i(\#) = \#$

This transformation can be extended with a transformation projecting the dynamic jumps and the conditional instruction to PGLDg, which can be subsequently translated into the lowest level: PGLA.

References

[1] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998.

[2] J.A. Bergstra and M.E. Loots. Projection semantics for flow charts. Technical Report 5, Faculty of Philosophy, Tilburg University, 1998. also ch. 3 of this volume.

[3] J.A. Bergstra and M.E. Loots. Abstract data type coprograms. Technical Report xx, Department of Philosophy, 1999. also ch. 4 of this volume.

[6] As a consequence the next thread to take a turn will be the newly created one.

Program aggregation and the user perspective

J.A. Bergstra* & M.E. Loots†

March 29, 1999

Abstract

Program aggregation turns a family of named programs into a conceptual unit. Program aggregates are natural from the perspective of a program user. Program aggregates play a three-fold role: as a semantic tool for the projection of object-oriented software, as a technique for programming coprograms on the basis of existing coprograms, and as a simple model for the concept of a software component.

1 Introduction

The user of programs will probably run several programs simultaneously, therefore needing a naming scheme. Running a program is done by producing the relevant program name in the right context. Thus, it seems plausible to regard calling a program by name as an action itself, the execution of the action being given as the execution of the underlying program. The position taken in [1] that programs (or rather program components) have no upper interface is debatable. The very name of a program may be considered the only inhabitant of an upper interface, the impact of its calling being identified with a complete execution of the program.

1.1 Program aggregates

The user perspective is now reflected in taking together a family of programs named by actions in the sense just discussed. Program aggregates formalize this mode of collection. Program aggregation is not just a program construction feature, aggregations not being programs. Nevertheless, some programs are semantically closer to aggregates than to individual programs. As a consequence we find that, for the purpose of projection semantics, some program notations can be projected onto a program aggregate much more naturally than to any particular low-level program. Program aggregates play a role in the understanding of the concept of a software component as well. The development

*University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl.

†Tilburg University, Faculty of Philosophy, email: M.E.Loots@kub.nl.

of a formalized concept of program aggregation is a first result of this paper. It is a necessay completion of the picture laid out by the development of program algebra and coprogram calculus.

A program aggregate fails to be a component due to the lack of explicit interface data. Adding interface details to a program aggregate creates the concept of a program aggregate component, providing a meaningful formalization of software components. Needless to say, program aggregates carry no state information, this marking their most prominent difference with coprograms. Formalizing the concept of software components in the setting of program algebra is a second result of the paper.

1.2 The coprogram construction operator

Seen as a product, program aggregates are software products. They fill a gap left by the problems involved in coprogram specification. Coprograms, as outlined in [1], require specification rather than construction. This poses a problem. We will explain the problem using a caricature: the principle of software imperialism (PSI). The principle reads as follows: everything of fundamental importance to software consists of software as well. Taking PSI seriously, for the sake of discussion, coprogram specifications constitute a problem. If coprograms are fundamental to software, they must be software. So either specifications are considered software themselves, an option of practical value, or specification must give way to program construction somehow. The role of program aggregates lies just here. This explanation of a role for program aggregate programming aimed at coprogram construction is a third result of the paper.

2 A program aggregation operator

Basic actions are taken from a parameter set Σ throughout this discussion. These actions are frequently provided with a structure. Only one kind of structure will be considered below: actions consisting of a name followed by zero or more parameter names, the parameters taking their values as natural numbers. We will write $(p_1, \ldots, p_n)$ for a parameter list, the absence of parameters usually being explicitly given by ().

2.1 Parameterized actions and programs

Although a discussion of program aggregates is technically feasible and meaningful if finite action collections and interfaces are considered, all applications will in practice involve the use of parameters. Limiting our attention to parameters ranging over NN (the set of natural numbers) or subsets thereof, we simplify the discussion at an acceptable loss of generality.

2.1.1 Parameterized actions and interface macros

Using parameters from $(p_0, \ldots)$, actions from Σ have the following forms: $a()$ and $a(p_1, \ldots, p_n)$. The use of parameterized actions implies the necessity of allowing parameterization for interface description as well. An interface macro is an expression of the form $a(-, \ldots, -)$ indicating the precise number of parameters for an action named a. The interface $\Sigma = \{a(), b7(-), aac(-), c3g(-,-)\}$ contains $a()$ together with all substitution instances $b7(n_1), aac(n_1), c3g(n_1, n_2)$ with natural numbers n_1 and n_2.

Interfaces are sets of actions, macros being used as an abbreviation at best. If only a subset of parameter values is admitted set-theoretic notation is used: for instance $\Sigma = \{a(14,2), a(22,77)\} \cup \{b5dd(-,-), bdd(-)\} \cup \{c(-,n) \mid 2 < n < 1.000.000\}$.

2.1.2 Programs with data parameters

Programs are written in some program notation, here referred to as PGLY. PGLY_Σ programs can have basic actions in Σ and it is natural to extend PGLY_Σ to $\mathrm{PGLY}_{\Sigma,p}$, allowing the use of formal parameters in basic actions throughout. A $\mathrm{PGLA}_{\Sigma,p}$ program may read as follows: $a(); (+b((p_1,p_2); !; \#; c((p_4)^\omega$, assuming $a(), b(-,-), c(-) \in \Sigma$.

2.2 Program aggregates

Given an interface Σ, and program notation $\mathrm{PGLY}_{\Sigma,p}$, as well as an interface Γ, a program aggregate (for $\mathrm{PGLY}_{\Sigma,p}$ based on Γ) is an I-indexed family X^{agg} of pairs of actions and programs, written

$$a_i(t_1^i, \ldots, t_{n_i}^i) = X_i$$

for $i \in \mathrm{I}$. The following conditions have to be met:

- for all $i \in \mathrm{I}$ and $j \in \mathrm{NN}$, either $t_j^i \in \mathrm{NN}$ or t_j^i is a variable p_k for some k.
- all data parameters p_k occuring in X must also occur in the list $t_1^i, \ldots, t_{n_i}^i$,
- all substitution instances of $a_i(t_1^i, \ldots, t_{n_i}^i)$ are contained in Γ.
- no substitution instance of a left-hand side occurs twice. (The family of pairs can be considered a function.)

The left-hand sides of these equations are called program names (sometimes: method names), the right-hand sides being called program bodies (sometimes method bodies). The equations themselves can be called defining identities of the program aggregate.

Program aggregates have no interfaces, although a pseudo upper interface can be defined as well as a pseudo lower interface. For the aggregate X^{agg}, the pseudo upper interface $\Sigma^{ps}_{upper}(X^{agg})$ consists of all (parameterless) substitution instances of program names of X^{agg}, the pseudo lower interface $\Sigma^{ps}_{lower}(X^{agg})$

being defined as the union of the pseudo lower interfaces of all right-hand sides after full substitution for all parameters.

2.3 Program aggregate components

A program aggregate component is a triple $X^{aggc} = < \Gamma_u, X^{agg}, \Sigma_l >$ with Γ_u an interface contained in $\Sigma^{ps}_{upper}(X^{agg})$ and Σ_l an interface extending $\Sigma^{ps}_{lower}(X^{agg})$.

Γu is the upper interface of X^{aggc}, Σ_l its lower interface. (In general for a program aggregate component X^{aggc}, $\Sigma_l(X^{aggc})$ denotes its lower interface and $\Gamma_u(X^{aggc})$ its upper interface.)

2.4 Program aggregation operator notation

In some cases it is helpful to allow a direct notation for program aggregates. This notation should not be considered a program notation, as it must be entirely independent of the constituent bodies. If the notation were considered a program notation, additional requirements on (the syntax used for) program bodies would be necessary. In the case of a finite set of defining identities, they can be combined by means of ';;' used as a semi-colon. No program notation is supposed to contain an empty program, so that this notation is sufficiently unique. It is reasonable to assume that ';;' is commutative, associative as well as idempotent. It simply collects its elements into a set, while avoiding the notations of set theory. As an example consider:

$$a() = e(); +f(); \#\#1; g(5); ; b(2) = \#; ; a(p_3) = g(p_3); +f(2); -h(); \#\#2; g(p_3)$$

If a program notation is designed with the purpose of expressing program aggregates, ';;' should not be used. Some more elaborate syntax will generally be needed, such as for instance:

$$\texttt{beginAgg}\{; a() = \{; e(); +f(); \#\#1; g(5); \}; b(2) = \{; \#; \};$$

$$a(p_3) = \{; g(p_3); +f(2) : -h(); \#\#2; g(p_3); \}\texttt{endAgg}$$

This notation requires proper use of braces in the various bodies, thus requiring a consistent overall language design. Having designed such a language and having to present its semantics, choosing a program aggregate (using ';;'as a separator) is plausible.

3 Software components

The concept of a software component[1], currently of rising importance, can be formalized by means of the terminology just developed. Of course, the practice of software components has many more aspects to offer than our formalization will cover. The following aspects are fundamental to software components:

[1] An excellent exposition of the subject of software components is found in [2].

- Explicit interfaces: we assume that interfaces are polarized, allowing a distinction between upper and lower interfaces.
- Low level code. (Program notations in the range PGLA, PGLB, PGLC, PGLD are certainly low level. For substantial code no human individual will either write, read or maintain code at that level.)
- The absence of state information. Software components do not encapsulate program states, nor do they contain any coprogram or coprograms used by the component.[2]
- Conceptual unity. The collected parts (in our formalization: the bodies) of a component should constitute a meaningful ensemble.

By formalizing a software component as a program aggregate component over, say PGLA, all of these criteria are met satisfactorily, except for the last one. The subject of conceptual unity evidently escapes the format of program algebra, being a problematic subject for formalization anyhow.[3]

Software components are considered important for the following reasons:

- program notation independence. Typically, having been compiled beforehand, software components are programming-language independent,[4]
- software components can be physically distributed independently of platform components,
- the owner of a software component never faces a maintenance problem, maintenance by means of code modification simply being impossible (or at least requiring unreasonable efforts, beginning with decompilation into some high-level notation),
- because of extreme standardization, software components realizing some particular functionality can be upgraded automatically and world-wide from a central location,
- world-wide component data bases, allowing search on the basis of sophisticated naming schemes are considered feasible,[5]
- software components represent a shift of focus: from re-using high-level program code to using low-level code,

[2] In our terminology states are wrapped in coprograms. A 'component' inseparable from its state, but still mobile, is called an agent.

[3] This formalization of a software component can be considered a 'result' of this paper. Although the mathematical content of the formalization is insignificant, it might facilitate a rigorous discussion of novel software methods and techniques.

[4] In return an equally unfortunate machine dependency may occur.

[5] Program libraries allowing search based on actual program contents have proved to be an illusion until now.

- software component technology captures the interface orientation of object-oriented programming without having to assimilate its ideological contents regarding 'natural modeling of the world',
- not having a state, software components completely separate software production from software usage, thereby greatly simplifying the use of mobile code,
- software components represent a format for a product-oriented software industry, rather than a service- and consultancy-oriented software industry.[6]

4 Aggregate programming of coprogram drivers

We will discuss the role of aggregate programming in coprogram construction. Following the considerations in [1] the presentation of coprograms will require specification rather than program construction. In view of PSI (half-seriously formulated above), a problem emerges: can specifications be considered software (i.e. programs) or not. If so, the jargon of specification can be rephrased after all; if not, PSI is clearly violated.[7] A simple solution is to exploit the observation that a 'program aggregate using a coprogram' is in turn (almost) a coprogram.[8]

4.1 Coprogram classification

A preparatory classification of coprograms is needed in order to see clearly which programming technique where to apply. Berlow coprograms are ordered from concrete (close to hardware) to abstract.

4.1.1 Hardware coprograms

Hardware coprograms describe the functionality of a piece of hardware. The hardware may be existent or in the process of being designed. Two subclasses can be distinguished:

- memory coprograms, denoting hardware for storage of data,
- function coprograms, denoting hardware computing special operations.

[6] The production of an aggregate program is considered programming.

[7] Coprograms may be considered superfluous, their role being diminished accordingly. Data structures and memory structures are functionally equivalent to coprograms, however, thus underlining the conceptual promincence of coprograms in our framework.

[8] It is necessary that the lower interface of the program aggregate is contained in the upper interface of the coprogram being part of the combination.

An important characteristic of hardware coprograms is the finiteness of their service interface.[9] In practice, hardware coprograms must have a finite state-space as well.[10]

4.1.2 Primitive hardware coprograms, and platform configurations

We will assume the existence of a library of primitive hardware coprogram descriptions. The elements of the library are indicated by name, these names beginning with `cpph`. We will not give explicit examples of such library entries here, noting that elementary ALU functionality, as well as direct address memory mechanisms should somehow be provided. It is assumed that the library consists of `cpph1,cpph2,..,cpph100`. A platform for programming will provide some of the available hardware resources, in particular multiplicity. This is captured by means of a platform configuration, named `cphpc.x`, formalized as a coprogram calculus expression of the form:

$$\texttt{cphpc.x} = \texttt{f1.cpph}k_1 \oplus \cdots \oplus \texttt{f1.cpph}k_n$$

4.1.3 Data structure coprograms

Data structure coprograms capture the behavior of a data structure supposedly realized by means of programming on the basis of a particular platform configuration.

Like hardware coprograms, data structure coprograms have a finite service interface. Data structure coprograms may be thought of as virtual hardware coprograms.

4.1.4 Abstract data type coprograms

Abstract data type coprograms describe the functionality of an abstact data type. Abstract datatype coprograms can have an infinite service interface. An abstract data type coprogram can be considered an abstract virtual hardware coprogram. [11]

4.2 Coprogram drivers

A coprogram driver is a software utility, providing the service of a coprogram at a higher level of abstraction. The clearest examples of coprogram drivers are found when a data structure coprogram must be realized on top of a particular

[9] in many cases instructons have parameters ranging over a bounded subset of the natural numbers, e.g. numbers with a binary representation of length 32 only.

[10] The theoretical literature contains a number of 'idealized' harware coprograms having an infinite state-space. An important example is the Turing Machine Tape here denoted with `cpphtmt`.

[11] Abstract because they can never be actually realized. Abstract data type coprograms help to provide intermediate descriptions that still have to be compiled (projected) in preparation of the actual execution on a machine.

hardware configuration coprogram. Data structure coprogram drivers have a finite interface. In principle, it is acceptable to omit the use of parameters.[12]

4.2.1 The coprogram driver language extension; 'y'

To facilitate the writing of coprogram drivers two new instructions are added to each of the languages in the program algebra language hierarchy. Let PGLZ be a program notation, then PGLZy extends PGLZ with these actions: `y(true)` and `y(false)`.

`y(true)` returns a value `true` to the caller of a program, `y(false)` returning false. If a program X in PGLZy is called, it returns *the last* boolean returned by means of an action `y(true)` or `y(false)` in X. If no such action has taken place an error (M) is returned. `y(true)` and `y(false)` are called yield instructions.

4.2.2 Coprogram drivers

Let Γ denote a finite action interface. A coprogram driver for service interface Γ relative to service interface Σ is a program aggregate component $< \Gamma, X^{agg}, \Sigma >$. The programs of the aggregate X^{agg} are to be written in PGLAy_{Σ}.

4.2.3 Datastructure coprogram construction

A coprogram construction is a quadruple $< \Gamma, X^{agg}, \Sigma, H >$ with H a platform configuration coprogram. The Quadriple is a combination of a driver and a memory. The driver translates requests in its service interface (Γ) to program executions over the underlying coprogram H.

The operational intuition of the coprogram construction is as follows: if an instruction in Γ is called (say m), the corresponding aggregate program body (say X_m) of X^{agg} is performed over H starting from the state (say H') that it was left in after previous computions (of course having taken place in reaction to previous calls for instructions in Σ except in the initial state).

The execution of the aggregate program body returns the last value having been produced by a yield instruction (`y(true)` or `false`), if any yield instruction was performed during the run of the body X_m, and returns an error (M) otherwise.

The next state of the used coprogram is computed as $X_m \bullet_L H'$.

4.3 Data structure coprogram emulation

Taking data structures to include the memory structures as well, and noticing that the definition of coprogram construction is not technically dependent on the assumption of H being a platform configuration coprogram, the notion of emulation between different data structures emerges as follows:

[12] In practice interface elements only differing in their parameter values are counted just once, thus allowing a measure on interfaces leading to manageable proportions. Interfaces with less than 10 instructions are currently advocated by software methodology experts.

For data structure coprograms H_1 and H_2, an emulation of H_1 on H_2 is a coprogram driver $< \Sigma_{u1}, X^{agg}, \Sigma_{H_2} >$ constructing H_1 from H_2:

$$H_1 = \mathtt{cpDr}(< \Sigma_u, X^{agg}, H_2 >)$$

(writing Σ_{u1} for $\Sigma_u(H_1)$.

4.3.1 Emulation reducibility for data structure coprograms

For data structure coprograms H_1 and H_2 we say that H_1 is emulation reducible to H_2, written $H_1 \leq_{emd} H_2$, if there is an emulation of H_1 on H_2.

If $H_1 \leq_{emd} H_2$ the emulation degree of H_2 is said to be at least as high as that of H_1. We list some facts concerning emulation reducibility:

- If only finite memory structures are considered, there is no data structure coprogram from which all other memory structures can be emulated.
- Given a finite state memory coprogram H, however, there is a natural number k such that H can be emulated from k disambiguated copies of the boolean register `cpphbr`. In symbols:

$$H \leq_{emd} \mathtt{f1.cpphbr} \oplus \cdots \oplus \mathtt{f}k\mathtt{.cpphbr}$$

- In case infinite memories are allowed, the Turing Machine Tape (`cpphtmt`) is such a maximal memory structure:

$$H \leq_{emd} \mathtt{cpphtmt}.$$

  For instance:

$$\mathtt{f1.cpphtmt} \oplus \mathtt{f2.cpphtmt} \leq_{emd} \mathtt{cpphtmt}.$$

- `cpphtmt` and, as a consequence, all other memory coprograms, can be emulated from two disambiguated copies of a natural number counter `cpphnc`:

$$\mathtt{cpphtmt} \leq_{emd} \mathtt{f1.cpphnc} \oplus \mathtt{f2.cpphnc}.$$

- Export potentially lowers the place of a datastructure coprogram in the emulation hierarachy:

$$\Sigma\Box H \leq_{emd} H.$$

4.3.2 Emulation equivalence and emulation degrees

We will write $H_1 \equiv_{emd} H_2$ if both $H_1 \leq_{emd} H_2$ and $H_2 \leq_{emd} H_2$ hold. Emulation equivalence, as we call $\equiv_{emd}$, is an equivalence relation on coprograms. Some elementary properties of emulation reducibility and emulation equivalence is helpful:

$$H_1 \leq_{emd} H_1 \oplus H_2 \ \mathtt{if}\ \Sigma_u(H_1) \cap \Sigma_u(H_2) = \emptyset$$

$$\mathtt{f}.H \equiv_{emd} H$$

$$H_1 \oplus H_2 \equiv_{emd} H_2 \oplus H_1$$

$$(H_1 \oplus H_2) \oplus H_3 \equiv_{emd} H_1 \oplus (H_2 \oplus H_3) \ \mathtt{if}\ \Sigma_u(H_1) \cap \Sigma_u(H_2) \cap \Sigma_u(H_3) = \emptyset$$

5 Conclusions

Program aggregation has been introduced and the concept of a software component has been formalized accordingly. Aggregate programming has been illustrated by means of the issue of coprogram driver programming for data structure coprograms.

References

[1] J.A. Bergstra and M.E. Loots. Abstract data type coprograms. Technical Report xx, Department of Philosophy, 1999. also ch. 4 of this volume.

[2] C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addisson-Wesley, The ACM-Press, New York, 1997.

Exercises on Programs, Interfaces and Components

J.A. Bergstra* & M.E.Loots†

March 30, 1999

Abstract

We provide three series of exercises belonging to 'Lectures on the logic of software complexiy'. The series are of different levels: a basic or mechanical level including a functional level, a conceptual level and a project level; in some cases exercises also provide new material.

1 Preliminary remarks

These exercises go together with: 'Programs, Interfaces and Components' (Eds. J.A.Bergstra & M.E.Loots). The exercises have different levels, the mechanical level being sufficient to get acquainted with the terminology and its use. The functional level is an extension of the mechanical level: it is called functional because is suggests thinking in terms of the required functionalities. The conceptual exercises aim at an enrichment of the conceptual backgound, the project level essentially providing suggestions for straightforward extensions of the story. Some preparatory comments are in order:

- Void actions are actions that by definition always return `true`. If an action is specified as void, additional information on its boolean yield is redundant.
- If an action is declared boolean-returning, it is explicitly non-void. Information about (or use of) returned boolean values can be expected.
- Three different forms of operation of a program on data will be distinguished: the input/output operator, the completed path operator and the completed stuttering free path operator.
- A coprojection is an embedding of a lower-level language into a higher-level one. Often projection/coprojection pairs are each other's inverse (or almost each other's inverse).

*University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl

†Faculty of Philosophy, Tilburg University, email: M.E.Loots@kub.nl

- We write PGLDci for PGLD with comment instructions in order to distinguish this 'c' representing comments (rather than conditions).

2 Notations for input and output

In order to clarify the exercises, a distinction is made between the various ways a program can operate on a particular input. We assume that a state space S is given, providing an effect function for each $a \in \Sigma$. For each $a \in \Sigma$ a yield function $y_a(-)$ is present as well. Effect functions and yield functions are considered part of the state space. If a number of state spaces occur simultaneously, they should be given distinctive names. We will use $F, F_1, F_2, \ldots$ as names for state spaces. IO-operators assign to programs (and behaviors) a mapping from input states to an appropriate kind of output, the kind of output being indicated in a superscript. In full generality an IO-operator has the state space (F) as a subscript, followed by its program notation. For instance:

$$X \bullet^{io}_{F,pgla} s$$

denotes the state computed by PGLA_Σ program X when started running over state space F from input state s. If the run of the program aborts, the expression denotes M; if the run diverges, it is equated with D.

2.1 Conventions for IO-operator decorations

Subscripts and superscripts for IO-operators are decorations, added only to avoid confusion. If just one IO-operator is needed, the superscript will often be dropped. For instance in 'Program algebra for component code' $- \bullet -$ refers to $- \bullet^{io} -$. Reading the previous statement the other way around the paper just mentioned provides a formal definition of $- \bullet^{io} -$.

If the context is clear about the state space in use, that subscript is often omitted as well. In most cases the following should work as a rule: $- \bullet^{\alpha} -$ stands for $- \bullet^{\alpha}_{F} -$ with α some superscript (e.g. io) and F a state space that has clearly been defined earlier in the text (in particular the last one).

If X has been introduced as a PGLZ[1] program before, it is standard convention to let $-\bullet^{\alpha}_{F,pglz}-$ be abbreviated by $-\bullet^{\alpha}_{F}-$ and $-\bullet^{\alpha}_{pglz}-$ by $-\bullet^{\alpha}-$ and so on.

Implicit use of behavioral meaning is acceptable too. If X is a PGLZ program then $X \bullet^{\alpha}_{F,pglz} s$ abbreviates the more orthodox $|X|_{pglz} \bullet^{\alpha}_{F} s$. If PGLZ can be inferred from the context this simplifies further to $X \bullet^{\alpha}_{F} s$ or $X \bullet^{\alpha} s$ or just $X \bullet s$.

[1] PGLZ serves as a metavariable over program notations.

2.2 Superscripts and output forms

Three output kinds will be distinguished.[2] This may seem to be pedantic but it pays off by providing a strong intuition.

input output tuples The superscript *io* indicates that the operator defines a mapping from state to state. The resulting state (in the case of convergence) is called the output.

completed paths *cpath* indicates that the output of a computation is a sequence of states starting with the input state, ending with the output state (in the sense of *io* mentioned above) and containing an intermediate state for every Σ action generated by the program. If no finite completed path exists the *cpath* semantics produces an infinite path. Completed paths can end in an error, the point where the error occurred being indicated precisely. Completed paths have the following forms:

- $s_1, \ldots, s_{k-1}, \surd$ (for $k \in$ NN, $s_i \in$ S), (correctly terminating path), the '$\surd$' having been added to affirm correct termination,
- $s_1, \ldots, s_{k-1}, M$ (path ending in an abort instruction)
- $s_1, \ldots, s_{k-1}, D$ (path ending in an internal divergence of the program)
- $s_1, \ldots, s_{k-1}, \ldots$ (infinite path)

completed stuttering-free paths Stuttering takes place if two or more consecutive states in a path coincide. This is likely to happen in the case of tests that do not change the state. The superscript *csfpath* indicates that a completed path must be produced with all repetitions removed. In a diverging computation (infinite path in the sense of *cpath*) the repetitions occurring in the 'stutter' may not be removed. This problem is solved by omitting the '$\surd$', thus finding another form of path:

$s_1, \ldots, s_{k-1}$ (infinite path, repeating s_{k-1} forever, written in stuttering-free form).

3 Mechanical level: software mechanics

3.1 Program algebra: PGLA

1. Determine the number of instructions of the following programs:
   (a) $\#11; !; \#4$
   (b) $!^\omega$
   (c) $-c; b^\omega$

[2] The ramification of output kinds increases significantly if non-deterministic state spaces are taken into account. A state space is non-deterministic if for some action and state the effect function provides a set of result states rather than a single one, the set having more than one element.

(d) $+a;!;b!$

(e) $!^{\omega};pred;!$

2. For each of the following program expressions determine the 3-th instruction:

   (a) $a;a;a;a;a;a$

   (b) $b;+c;b$

   (c) $b;+c$

   (d) $(b;+c)^{\omega}$

3. For each of the following program expressions determine the 5-th instruction:

   (a) $(b;+c)^{\omega};b$

   (b) $-b$

   (c) $!;!$

4. In the cases below in each exercise there is a set of 2 or 3 equations that defines a program X. The question is to provide a single closed term that represents X. (I.e. to solve the system of equations. The point of these questions is to get the brackets right).

   (a) $X = b;Y;c$ `where` $Y = !;+f;!;c$

   (b) $X = b;Y^{\omega};c$ `where` $Y = !$

   (c) $X = a;Y^{\omega}$ `where` $Y = b;+c;Z;!$ `and` $Z = b;b$

5. In each of the following cases determine for all k:

   (a) $i_k(a;b;(c;a;b)^{\omega};!;b)$

   (b) $i_k((a;b;c)^{\omega};!;a)$

   (c) $i_k(a;f;(c;a;b)^{\omega};!;b)$

   (d) $i_k(a;(b;f;a);b^{\omega};!;a)$

6. Show that for all k the following identities hold:

   (a) $i_k((a;+b;c)^{\omega}) = i_k(a;+b;c;(a;+b;c)^{\omega})$

   (b) $i_k((a;+b;c)^{\omega}) = i_k(a;(+b;c;a)^{\omega});!$

   (c) $i_k(\#250;+b;c;(a;!;\#2;+b;c)^{\omega}) = i_k(\#250;(+b;c;a;!;\#2)^{\omega})$

3.2 Calculation on the linear integer grid

In these exercises we assume a deterministic state space (transition system) F_{int} with the integers as its domain. The void actions are succ (with the effect of increasing the number that determines the state by one) and pred (which decreases the state coordinate by one). The boolean actions are pos, neg, zero. Each of these has the identity function as its effect (that is no change in state). The yield functions are as follows $y_{pos}(z) = T$ if z is positive and $y_{pos}(z) = F$ otherwise. Similar definitions are issued for neg and zero.

1. Determine the number of instructions of the following programs:
   (a) $\#11; +pos; \#3; -neg$
   (b) $!^\omega; pred$
   (c) $-zero; succ^\omega$
2. Find for the following program expressions an expression in first canonical form which is equal to it as a program object (and which can be proven equal by means of the five program object equations). Indicate how the equivalence proofs work.
   (a) $!^\omega; !$
   (b) $(!;!)^\omega; (!;!)^\omega; !$
   (c) $succ; (+neg; !)^\omega; pred; !$
   (d) $succ; (+neg; !; (pred; succ)^\omega)^\omega; (pred; !)^\omega$
   (e) $succ; (-neg; !; (pred; +pos^\omega; pred)^\omega)^\omega; (succ; -zero)$
3. For each of the following program expressions determine the second instruction:
   (a) $+zero$
   (b) $!; pred; succ$
   (c) $\#2; +zero; pred$
4. Transform the following program expressions in first canonical form to PGA-equivalent expressions in second canonical form:
   (a) $\#5; (succ; !)^\omega$
   (b) $\#5; +zero; (pred; !)^\omega$
   (c) $\#8; +zero; !; (\#7; succ; pred; !)^\omega$
   (d) $\#15; +neg; !; (\#14; \#3; \#2; !; succ; +zero; !)^\omega$
5. Compute the following behaviors:
   (a) $|pred; !; succ; !; pred|$
   (b) $|!; !; pred^\omega|$

(c) $|\#2; !; succ; pred; !|$

(d) $|+zero; !; succ; -zero!; pred|$

6. Are the following behavioral equivalences valid (writing $|X|$ for $|X|_{pgla}$)?

   (a) $!; !; ! \equiv_{be} !; !$

   (b) $!; succ \equiv_{be} !; pred$

   (c) $\#2; !; pred; succ \equiv_{be} \#3; +neg; -pos; succ; pred$

   (d) $+zero; !; succ; ! \equiv_{be} -zero; \#3; succ; !; !$

7. Determine the following values:

   (a) $|succ; pred; \#1; !| \bullet^{io} 5$,

   (b) $|(+zero; !; pred)^{\omega}| \bullet^{io} 4$,

   (c) $|(+zero; !; succ)^{\omega}| \bullet^{io} -3$,

   (d) $|pred. - neg; !; succ; (-zero; \#2; !; pred; pred)^{\omega}| \bullet^{io} 5$.

   (e) $|succ; -neg; !; pred; (-zero; \#2; !; pred; pred)^{\omega}| \bullet^{io} -1$.

   (f) $|succ; -pos; !; pred; (-zero; \#2; !; succ; succ; succ)^{\omega}| \bullet^{io} -1$.

8. Are the following input-output equivalences valid (relative to F_{int}):

   (a) $succ; !; ! \equiv_{io} succ; !; pred$,

   (b) $! \equiv_{io} succ; pred; !$,

   (c) $+zero; !; pred \equiv_{io} -zero; pred; !$,

3.3 Calculation on the 2D integer grid

In this series of exercises we consider computation on the two dimensional integer grid. Elements of the grid are pairs $< x, y >$ with x and y integers. In the exercises below the state will be a point $< x, y >$ in the 2D integer grid which is transformed by the void actions and inspected by the boolean actions.

The void actions are mup (move up, or increment y by 1), mdown (move down, or decrement y by 1), mleft (decrement$x$ by 1), mright (increment x by 1). The boolean-returning actions, none of them changing the state, are xpos (returning `true` if $x > 0$), xnull (returning `true` if $x = 0$), xneg (returning `true` if $x < 0$), ypos, ynull, yneg (returning `true` if respectively $y > 0$, $y = 0$, $y < 0$) and diag (returning `true` if $x = y$) and codiag (returning `true` if $x = -y$). Of course the boolean-returning actions will return `false` whenever `true` is not returned.

The state space outlined above is named G2D. It will play an important role in several exercises below.

1. Find for the following program expressions an expression in first canonical form which is equal to it as a program object (and which can be proven equal by means of the three program object equations). Indicate how the equivalence proofs work.

(a) $!^{\omega};!$

(b) $mdown;(+ypos;\#2;!;mup;mdown)^{\omega};xneg;zpos;!$

(c) $mright;(+ynull;!;(-ynull;mdown;+ypos)^{\omega});(+znull;!)^{\omega}$

2. For each of the program expressions in the previous exercise determine:

(a) the 2-nd instruction,

(b) the 10-th instruction,

(c) the 1.025.465-th instruction.

3. Compute the following behaviors:

(a) $|mup;!;mleft;!;mdown|$

(b) $|!;mdown;\#5;mup^{\omega}|$

(c) $|\#2;!;+yneg;-xpos;!;mup|$

(d) $|+xnull;!;-xneg;mup;!;mdown|$

4. Are the following behavioral equivalences valid?

(a) $!;! \equiv_{be} !;!;!;!$

(b) $!;mdown \equiv_{be} !;mleft$

(c) $\#2;!;mdown;mup \equiv_{be} \#3;+xneg;-xpos;mup;mdown$

5. Determine the following values:

(a) $|mup^{\omega}|\bullet^{io} < 0,0 >$,

(b) $|mleft;mright;\#1;!|\bullet^{io} < 0,5 >$,

(c) $|(+xnull;!;mdown)^{\omega}|\bullet^{io} < 4,7 >$,

(d) $|(+diag;!;mup;-codiag;!)^{\omega}|\bullet^{io} < -3,-7 >$,

6. Determine the following completed paths:

(a) $|\#2;!;!|\bullet^{cpath} < 2,3 >$

(b) $|\#1;mup;mleft;!|\bullet^{cpath} < -2,-3 >$

(c) $|\#1;mup;+xnull;mleft;!|\bullet^{cpath} < -2,-3 >$

(d) $|\#1;mup;+xnull;-ynull;mleft;!|\bullet^{cpath} < -2,-3 >$

(e) $|\#1;mup;\#3;+xnull;-ynull;mleft;!|\bullet^{cpath} < -2,-3 >$

7. Determine the following completed stuttering-free paths:

(a) $|\#2;!;!|\bullet^{csfpath} < 2,3 >$

(b) $|\#1;mup;mleft;!|\bullet^{csfpath} < -2,-3 >$

(c) $|\#1;mup;+xnull;mleft;!|\bullet^{csfpath} < -2,-3 >$

(d) $|\#1;mup;+xnull;-ynull;mleft;!|\bullet^{csfpath} < -2,-3 >$

(e) $|\#1; mup; \#3; +xnull; -ynull; mleft; !|^{\bullet^{csfpath}} < -2, -3 >$

8. Determine the following completed paths:

   (a) $|mleft^{\omega}|^{\bullet^{cpath}} < 0, 0 >$

   (b) $|(mup; mdown)^{\omega}|^{\bullet^{cpath}} < -1, -1 >$

   (c) $|(mup; mright; mdown; mleft)^{\omega}|^{\bullet^{cpath}} < -1, -1 >$

   (d) $|(mup; mright; mdown; mright)^{\omega}|^{\bullet^{cpath}} < 0, 0 >$

   (e) $|(mup; mright; +ynull; mdown; mright)^{\omega}|^{\bullet^{cpath}} < 0, 0 >$

9. Determine the following completed stuttering-free paths:

   (a) $|(mleft; -xnull)^{\omega}|^{\bullet^{csfpath}} < 0, 0 >$

   (b) $|(mup; +xnull; \#1; mdown)^{\omega}|^{\bullet^{csfpath}} < -1, -1 >$

10. Determine the values as mentioned in exercise 5 with $\bullet^{io}$ replaced by $\bullet^{cpath}$ and by $\bullet^{csfpath}$.

3.4 PGLB

In this exercise the setting for computing on the linear grid is used as laid out in paragraph 3.2. Programs are now written in PGLB instead of PGLA.

1. Determine the number of instructions of the following programs:

   (a) $succ; pred; \backslash\#2; -zero; !; !$

   (b) $!\backslash\#1; pred$

   (c) $-zero; succ; \backslash\#1$

2. For each of the following program expressions determine the second instruction:

   (a) $\backslash\#2; +zero$

   (b) $!; \#2; pred; succ$

   (c) $\#2; !$

3. Compute the following behaviors:

   (a) $|pred; !; \backslash\#0; !; pred|$

   (b) $|\#3; !; !; \backslash\#1; pred; \backslash\#1|$

4. Are the following behavioral equivalences valid?

   (a) $!; \backslash\#2; ! \equiv_{be} \#3; !; \backslash\#1$

   (b) $succ; !; ! \equiv_{be} \#2; succ; !$

   (c) $\#2; !; pred \equiv_{be} \#2; \backslash\#5; pred$

5. Determine the following values:

   (a) $|succ; pred; \#1; !| \bullet^{io} 5$,

   (b) $| + zero; !; pred; \backslash\#3| \bullet^{io} 4$,

   (c) $| + zero; !; succ; \backslash\#3| \bullet^{io} -3$,

   (d) $|pred. - neg; !; succ; -zero; \#2; !; pred; pred; \backslash\#5| \bullet^{io} 5$.

   (e) $|succ; -neg; !; pred; -zero; \#2; !; pred; pred; \backslash\#5| \bullet^{io} -1$.

6. Are the following input-output equivalences valid:

   (a) $succ; !; ! \equiv_{ioF_{int}} succ; !; pred$,

   (b) $! \equiv_{ioF_{int}} succ; pred; \backslash\#2$,

3.5 PGLA projections for PGLB

1. For the following PGLA programs carry out these two transformations in succession: `pgla2pglb` and `pglb2pgla`:

   (a) a

   (b) $a; \#$

   (c) $a; !$

   (d) $-f; !; c; a; !$

   (e) $+f; c^{\omega}$

2. Determine the 5-th approximation of $| - |_{pglb}$ for the following PGLB programs:

   (a) a

   (b) $a; !$

   (c) $a; \#$

   (d) $a; +b; \backslash\#2; !$

   (e) $a; a; a;$

   (f) $a; a; a; a; a; a; a$

3.6 PGLC

1. Apply `pglc2pglb` followed by `pglb2pgla` to the following programs:

   (a) $a; \backslash\#1; +g; +h; \backslash\#10$

   (b) $\backslash\#2; \backslash\#0; +g; +h; \backslash\#2$

   (c) $\#0; -c; \backslash 1\#2; +g; f; +h; \backslash\#2$

2. For the following programs in PGLA carry out these four transformations in succession: `pgla2pglb`, `pglb2pglc`, `pglc2pglb` and `pglb2pgla`:

(a) $\#$

(b) $!$

(c) $+g$

(d) $a; !^{\omega}$

(e) $a; \#^{\omega}$

3. Determine the 4-th approximation of $|-|_{pglc}$ for the following PGLC programs:

   (a) $+g; \#; c$

   (b) $+g; \backslash\#1; \#1; c$

   (c) $a; b; c$

   (d) $a; b; c; b; a$

3.7 PGLD

1. Determine the 3-rd approximation of $|-|_{pgld}$ for the following PGLD programs:

   (a) $\#\#10; a; \#\#2; b$

   (b) $\#\#3; a; b; +f; d$

   (c) $a; b; c$

   (d) $\#\#2; \#; \#\#1; -f$

2. For the following programs in PGLD carry out these six transformations in succession: `pgld2pglc`, `pglc2pglb`, `pglb2pgla`, `pgla2pglb`, `pglb2pglc`, and `pglc2pgld`:

   (a) a

   (b) $\#$

   (c) $\#\#1$

   (d) $\#\#2$

   (e) $+g; -f; \#\#4; +f\#\#1; c$

3. Provide an informal argument why the following PGLD programs have a pairwise identical behavior:

   (a) $h; \#\#3$ and $+h; \#\#2$

   (b) $a; \#\#3; c; \#\#2$ and $a; c; \#\#2$

   (c) $a; +b; c; \#\#5; -g; \#\#2; +f; \#\#7$ and $a; +b; c; -g; \#\#2; +f; \#\#6$

   (d) $b; +f; \#\#5; \#\#7; d; e; +g; \#\#1; \#\#10; -g; \#\#12$ and
   $b; -f; \#\#7; \#\#5; d; e; -g; \#\#10; \#\#1; +g; \#\#12$

3.8 Program algebra with comments: PGLDc, PGLDcit

1. Are the following PGLDci programs ICCC (instruction counter correct):
   (a) $a; b; c; d; e; f; g; \#\#3; b; v; \#\#18; \%\,\mathtt{ic}\,11; f; f$
   (b) $a; +b; -c; \%\,\mathtt{ic}\,4; +d; e+; f; g; \#\#3; b; v; \#\#18; \%\,\mathtt{ic}\,13; f; f; f; f$
2. Transform the following PGLDci templates into ICCC PGLDci programs (of course it suffices to determine the correct values for the jump counter variables $x1$ and $x2$):
   (a) $a; b; c; d; \#\# x1; e; f; g; \#\#3; b; v; \#\#18; \%\,\mathtt{ic}\,x1; f; f$
   (b) $a; +b; -c; \%\,\mathtt{ic}\,x1; +d; e+; f; g; \#\#3; b; v; \#\#18; \%\,\mathtt{ic}\,x2; +f; \#\# x2; \#\# x1$

3.9 Dataflow and PGLDg, PGLDgc

1. Apply the coprojection `pgld2pgldg` to the following programs:
   (a) $+a; \#\#1; b$
   (b) $+b; \#\#7; \#\#1; c$
2. Apply the projection `pgldg2pgld` to the following programs:
   (a) $£2; c; \#\#£3$
   (b) $£2; +c; \#\#£2; -f; £3; £2$
3. Apply the projection `pgldgc2pgldg` to the following programs:
   (a) $\mathtt{if} + b\{; c; \}\{; d; \}$
   (b) $a; \mathtt{if} - b\{; +c(1); c(2); \}\{; -d(1); d(2); \}; e$
4. Determine the `pgldgc2pgldg` projection for the following PGLDGc programs:
   (a) $£5; a; \mathtt{if} - c\{; a; \#\#£5; \}\{; e; f; \}; g$
   (b) $£5; a; \mathtt{if} - c\{; a; \#\#£5; \}\{; e; f; g$
5. The program notation PGLSc[3] is the subset of PGLDgc where labels labels and goto's are omitted, and absolute jumps are not allowed either. The description of `pgldgc2pgldg` can be simplified into one that works from PGLSc to PGLDg. Explain how this simplification may work. Check the validity of the simplification with the above examples.
6. Define a projection from PGLSc to PGLA.
7. Define a reasonably precise set of rules on how to make two-dimensional pictures of the kind of data flow networks that have been projected onto PGLDg. Give two examples (each having three nodes at least, including one test node). Describe the translation of these examples into pGLDg.

[3]PGLS denotes the straight line programs, PGLSc extends that with a conditional construct.

3.10 PGLDg programming on G2D(p, q, r)

We will extensively return to the state space G2D consisting of the 2 dimensional grid as introduced in 3.3. That state space describes the motion of a single point in the plane under control of a program. We will extend the context by considering the motion of three independent points. The points are named p, q, r. The actions are become all prefixed with $p.$, $q.$ or with $r.$. An action $p.mup$ indicates to move up p by one. An action $q. + ynull$ tests whether the y-co-ordinate of q equals zero.

The state space thus obtained is named G2D(p, q, r). Its states ar indicated as follows: $< p(1,4), q(2,5), r(4,-1) >$ denotes a state with p at co-ordinates (1,4) etc. We allow omitting one or two points, the non-listed points being placed by default at co-ordinates (0,0). So the x-co-ordinate of p in $< q(2,-7) >$ equals 0.

1. We define some programs first for use in several exercises below, using w as a name ranging over $\{p, q, r\}$:

$$X_w = w.mup; w.mright; w.mup; w.mright,$$

$$Y_w = w.down; w.mright; w.down; w.mright$$

   Determine the following completed paths:

   (a) $\pounds 0; X_p; Y_p; \#\#\pounds 0 \bullet_{pgld}^{cpath} < q(1,1) >$

   (b) $\pounds 0; X_p; Y_p; X_q; Y_q; \#\#\pounds 0 \bullet_{pgld}^{cpath} < p(0,+5), q(0,-5) >$

   (c) $\pounds 0; X_p; Y_p; +p.xnull; X_q; Y_q; -q.xnull; \#\#\pounds 0 \bullet_{pgld}^{cpath} < p(0,+5), q(0,-5) >$

2. Find a description of the following completed (in fact infinite) paths:

   (a) $\pounds 0; p.mup; p.right; p.mdown; p.mleft;$
   $q.mright; q.mup; q.mright; q.mup; \#\#\pounds 0$
   $\bullet_{pgld}^{cpath} < p(0,10), q(0,-5) >$

   (b) $\pounds 0; p.mup; p.right; p.mdown; p.mleft;$
   $q.mup; q.right; q.mdown; q.mright; \#\#\pounds 0$
   $\bullet_{pgld}^{cpath} < p(0,10), q(0,-5) >$

3. Find a description of the following stuttering-free paths (hint: this may require some informal explanation, based on a 'graphical insight'):

   (a) $\pounds 0; p.mup; p.mdown; +p.ynull; p.mdown;$
   $q.mright; q.mup; q.mup; +q.diag; q.mright; \#\#\pounds 0$
   $\bullet_{pgld}^{csfpath} < p(0,10), q(0,-5) >$

   (b) $\pounds 0; p.mup; p.mdown; +p.ynull; \#\#\pounds 1;$
   $q.mright; q.mup; q.mup; \pounds 1; +q.diag; q.mright; \#\#\pounds 0$
   $\bullet_{pgld}^{csfpath} < p(0,10), q(0,-5) >$

3.11 Interfaces

1. Determine the pseudo lower interface of the following PGLDg programs:
   (a) $a(); +a(); \#\#£5; b(23); +b(1); £5; b(2)$
   (b) $£1;\ a(); +a(); \#\#£1; a(0); b(24); +b(18); £1; b(22)$
2. Determine the pseudo lower interface of the behavior of following PGLDg programs:
   (a) $a(); a(); \#\#£5; b(23); +b(1); £5; b(2)$
   (b) $a(); +a(); \#\#£5; b(23); +b(1); £5; b(2)$
   (c) $a(); +a(); \#\#£5; b(23); \#\#£5; +b(1); £5; b(2)$
   (d) $a(); +a(); \#\#£5; b(23); \#\#£5; +b(1); £5; b(1)$
   (e) $£1;\ a(); +a(); \#\#£1; a(0); b(24); +b(18); £1; b(22)$
3. Determine the upper interface of the following coprograms:
   (a) $\texttt{cpA1} = \texttt{g2.cpbr} \oplus \texttt{g4.cpbr} \oplus \texttt{g7.cpbr}$
   (b) $\texttt{cpA2} = \texttt{g1.cpbr} \oplus \texttt{g2.cpbr} \oplus \texttt{g3.cpbr} \oplus \texttt{g1.cpbr} \oplus \texttt{g2.cpbr}$
   (c) $\texttt{cpA3} = (\Sigma_s(\texttt{cpA2})\Box(\texttt{g1.cpbr} \oplus \texttt{g2.cpbr} \oplus \texttt{g3.cpbr})) \oplus \texttt{g1.cpbr} \oplus \texttt{g2.cpbr}$
   (d) $\texttt{cpA4} = (\Sigma_s(\texttt{cpA1})\Box(\texttt{g1.cpbr} \oplus \texttt{g2.cpbr} \oplus \texttt{g3.cpbr})) \oplus \texttt{g1.cpbr} \oplus \texttt{g2.cpbr}$
4. Determine the following interfaces:
   (a) $\Sigma_l^{ps}(X_1)$ with
   $X_1 = \texttt{g1.setc(true)};\texttt{+g3.eqc(false;\#\#£3;g3.setc(false))}$
   (b) $\Sigma_l^{ps}(|X_2|_{pgldg})$ with
   $X_2 = \texttt{g1.setc(true)};\texttt{+g2.eqc(false);\#\#£3;g2.setc(false)}$
5. In each of the following cases decide whether the application of the use-operator is permitted; if so determine the signature mentioned (reference is made to the coprogram definitions above).
   (a) $\Sigma_l(|X_1/\texttt{cpA1}|_{pgldg})$
   (b) $\Sigma_l(|X_3/\texttt{cpA2}|_{pgldg})$
   (c) $\Sigma_l(|X_3/\texttt{cpA4}|_{pgldg})$
6. In each of the following cases decide whether the occurrence of the apply-operator is permitted:
   (a) $X_3 \bullet \texttt{cpA2}$
   (b) $X_2 \bullet \texttt{cpA3}$
   (c) $X_1 \bullet \texttt{cpA4}$

7. For a setting with actions having a single NN parameter we define that an interface is a collection Σ of actions $u(j)$ such that if $j_1 < j_2 < j_3$ and $u(j_1) \in \Sigma$ as well as $u(j_3) \in \Sigma$ then also $u(j_2) \in \Sigma$. Determine the smallest interface Σ such that $< \Sigma, X >$ is a program component for each of the following programs (simultaneously):

   - $a(7); a(2); \#\#\pounds 5; b(83); +b(1); \pounds 5; b(2)$
   - $a(8); +a(5); \#\#\pounds 5; b(23); +b(1); \pounds 5; b(2)$
   - $a(8); +a(6); \#\#\pounds 5; b(23); \#\#\pounds 5; +c(5); \pounds 5; b(2)$
   - $a(8); +a(2); \#\#\pounds 5; b(23); \#\#\pounds 5; +b(1); \pounds 5; b(1)$

8. Just like coprograms, state spaces have an upper interface. The key difference is that with coprograms the only possibility to change its state is via the upper interface, a state space in general being a 'device' rather than a 'memory' thereby subject to external influences as well. Clearly a coprogram can be considered a state space rightaway.

   Determine the upper interfaces of G2D (see 3.3) and (G2D(p,q,r) (see 3.10).

3.12 Recursion, PGLDg$_\mu$, PGLDgr

1. Apply `pgldgr2pgldgm` to the following programs:

   (a) $b; \pounds 2; \rho; +c; a; \rho\#\#\pounds 2; \pounds 3;$

   (b) $b; \pounds 2; -d; \#\#\pounds 3; \rho; +c; b; \rho\#\#\pounds 2; \pounds 3; c$

2. Determine the third approximation of the following behaviors:

   (a) $|a; \pounds 0; +b; \rho\#\#\pounds 0|_{pgldgr}$

   (b) $|a; \pounds 0; +b; \rho\#\#\pounds 0; \rho; c|_{pgldgr}$

   (c) Analyse the input-output behavior of the following program (moving only point p):

      i. $+p.neg; \#\#\pounds 3;$
      $\pounds 0; +p.ynull; \#\#\pounds 1; p.mdown; p.mright; \rho\#\#\pounds 0;$
      $p.mdown; p.mright;$
      $\pounds 1;$
      ρ

      (Hint: try some computations on inputs $p(i,j)$ with small j. Perform the computations directly on the basis of an intuitive operational understanding of the advanced control actions for recursion.)

      ii. $+p.neg; \#\#\pounds 3;$
      $\pounds 0; +p.ynull; \#\#\pounds 1; p.mdown; p.mright; p.mright; \rho\#\#\pounds 0;$
      $p.mdown; p.mdown; p.mleft; p.left; p.left;$

$\pounds 1;$
ρ

(d) Determine the following (infinite) stuttering free path:
$(\pounds 4;$
$+p.neg; \#\#\pounds 3;$
$\pounds 0; +p.ynull; \#\#\pounds 1; p.mdown; p.mright; \rho\#\#\pounds 0;$
$p.mdown; p.mright;$
$\pounds 1;$
$\rho;$
$\pounds 3; +p.diag : \#\#\pounds 4; p.mup; \#\#\pounds 3)$
$\bullet^{csfpath}_{pgldgr} < p(2,1) >$

3.13 Multi-threading, PGLDgmt

1. Apply `pgldgmt2pgldgmu` to $a; \texttt{fT}\#\#\pounds 0; b; \pounds 0; c$.

2. We will once more considered computation over G2D(p, q, r). We will define some programs first, w ranging over the three-point names:

   $X(w) = w.mup; w.mleft; w.mdown; w.mright,$
   $Y(w) = w.mup; w.mleft; w.mdown; w.mright; w.mright,$

   Evaluate the following expressions[4]:

   (a) $(\texttt{fT}\#\#\pounds 0; X(p); \#\#\pounds 1; \pounds 0; Y(q))$
   $\bullet^{io}_{pgldgmt} < p(7,7), q(3,3) >$

   (b) $(\texttt{fT}\#\#\pounds 0; X(p); Y(p); Y(p); \#\#\pounds 1; \pounds 0; Y(q); Y(q); Y(q); Y(q))$
   $\bullet^{io}_{pgldgmt} < p(7,7), q(3,3) >$

3. Give a description of the following path:
   $(\texttt{fT}\#\#\pounds 0; \pounds 1; X(p); Y(p); Y(p); \#\#\pounds 1;$
   $\pounds 0; Y(q); Y(q); Y(q); Y(q); \#\#\pounds 0)$
   $\bullet^{io}_{pgldgmt} < p(7,7), q(3,3) >$

   (Hint: look for some 'qualitative' description only.)

4. Are you able to determine completed (stuttering-free) paths in any of the above four cases?

[4] Using an intuitive understanding of the multi-threading primitives

4 Functional level

4.1 Programming point movements in the 2D grid, with PGLDci

1. Consider the state space of excercice 3.3. Find 4 (possibly different) PGLDci programs X such that:

   (a) $|X|\bullet^{io} < 100, 0 >=< 0, 100 >$

   (b) $|X|\bullet^{io} < 150, 150 >=< 150, -151 >$

   (c) $|X|\bullet^{io} < 153, 148 >=< 151, -147 >$

   (d) $|X|\bullet^{io} < 1000, 0 >=< 1000, 1000 >$

2. Find a PGLDci program X such that:

   - $|X|\bullet^{io} < 50, -50 >=< -50, 50 >$ and
   - $|X|\bullet^{io} < 85, 85 >=< -85, -85 >$

3. We add an action wait. The effect of wait is that it leaves the state where it is. If we regard a running program as a point in motion on the 2D plane performing just one step (action) per time unit, wait allows it to stand still on the spot.

   Find a program X that moves from point $< 0, 1000 >$ (input) to point $< 1, 1000 >$ in approximately 5000 steps.

4.2 Programming point movements with gotos and labels

1. A 2D grid with an inaccessible subarea.
   We assume that the collection $IN = \{< x + 1, y > | x = y \& x > 0\}$ is inaccessible in the sense that every motion which brings the state at a point in IN constitutes an error showing up up by resetting the state to $< 0, 0 >$. The reset dumps all memory that was contained in the state.

   (a) Find a program X such that $|X|\bullet^{io} < 102, 100 >=< 100, 102 >$. (We say that the program moves from $< 102, 100 >$ to $< 100, 102 >$. This is ad hoc jargon for this geometric kind of setting.)

   (b) Find a program that moves from $< 102, 100 >$ to $< 100, 102 >$ in twice as many steps as the program found in the previous exercice.

   (c) We add four boolean-returning actions that do not change the state: accup, accdown, accleft, accright, acc stands for 'accessible' and the actions return `true` if the gridpoint indicated by the tail of the action name is accessible. Using these sensor conditions a program can perform a motion in circumstances where it has incomplete knowledge of the inaccessible area. The question now is to produce a program that performs the same motion as above, but, that one can deal with the possibility that instead of the (half) diagonal in the (pos,pos)

quadrant, the half diagonal in the (neg neg) quadrant is not accessible.

5 Conceptual level

5.1 System calls as program actions I

5.1.1 Introduction

The methodology of program algebra suggests regarding a large number of actions as advanced control instructions. Projection semantics expands these to low -level actions (finally at PGLA level) in many cases involving actions for the reply co-service interface along which a program uses an appropriate coprogram. Quite at the opposite end of advanced control commands are the system calls. They instruct the run time system to perform certain moves that could not possibly have been programmed by means of a PGLA program, the latter having no access[5] to such system call actions. We will now investigate some options for system calls.

All system calls have their name prefixed with `sca`, albeit that OO-notation may lead to other prefixes in addition. For instance: if g is the name of a file then g.`scaEmpty` will denote an action that tests the file for being empty. The reason for having the file g under the responsibility of 'the system' rather than wrapping it in a coprogram is simple. Other programs may need to inspect the file as well. In addition the file will survive program termination. In this exercise we will take an informal attitude. Although formalization of the results is feasible, it would obscure the presentation. The discussions below will be entirely based on PGLA programs. That suffices in view of projection semantics for many other program notations. To simplify matters we will assume that programs do not use any coprograms.

System calls are processed by 'the system'. The system may be considered almost omniscient. In particular it has complete insight into the exact state of all running programs at any time, in the contents of all files and in the text of all programs. In addition the system is assumed to be extremely intelligent: it can answer any question posed in set theory. (A creative subject in a classical world.) No assumption of computability restricts the operation of the system.

5.1.2 Impossibility of a run-time halting information service

We will investigate whether the system can provide a run-time halting information service in the following sense:

A (reliable) run-time halting information service consists of a system call `scaHi` which makes no change to the system. If a program X calls `scaHi`, the system will reply `true` if X will terminate and it will reply `false` if execution of X will diverge (i.e. otherwise).

[5] Having no acces: making no use of.

We assume that 'the system' provides a correct halting information service. Consider the following program $X = +\texttt{scaHi}; \#0; !.$

1. Assume that X halts (its execution will terminate): check the following steps;
   - the call `scaHi` returns `true`,
   - upon which X runs into a diverging loop, whence it fails to halt!
2. Assume that X diverges (its execution will not terminate): check the following steps;
   - the call `scaHi` returns `false`,
   - upon which X terminates, whence it halts after all.
3. There is a contradiction. The existence of a runt-time halting service is highly implausible. The assumption that such a service should exist can be denied in two possible ways; which one do you prefer:
   (a) the very concept of halting with the strict dichotomy between 'halting' and 'not halting' is flawed (in the given circumstances),
   (b) the concept of halting is adequate, but the service cannot be offered.
4. Although the halting problem for PGLA programs seems to be easily solvable, this appears to be of no help at all. Which is the clearest explanation for that failure:
   (a) Halting is not clearly defined for PGLA programs using reply values for their actions. Clear notions are: uniform halting (halting along each execution path), and possible halting: (halting along some execution path, as well as uniform and possible divergence.) It is better to provide a 'possible halting information service' instead.
   (b) Asking questions about halting for a running system is asking for trouble: the circularity is too obvious!

5.2 System calls as program actions II

5.2.1 File management primitives

In order to handle aspects of batch processing some preparatory assumptions are needed. We will assume the existence of a file management system. Here are some further properties:

file contents. Files can contain programs (PGLA program objects), our current interest being restricted to files that can only contain PGLA programs (program objects).

file names. We will use $g, g_1, g_2, \ldots$ for file names. They may well be taken to denote natural numbers. After having been declared by means of an instruction in the program, file names are local.

file declaration. `scaNewFile`g = `empty` makes the system produce a new and empty file accessible to the declaring program under name g

emptiness test. The test instruction g`.scaFileEmpty()` tells its caller whether the file denoted with g is empty.

editing commands. Editing a file can be done by a program by means of editing instructions. The editing instructions are:

- `scaBgEdit`(u) begins an edit session by clearing the file and initializing it with a program consisting only of the instruction u.[6]
- `scaRep` replaces the program X stored in a file by X^ω. If the file is empty it remains so.
- `scaPrefix`(u) prefixes the program with instruction u.
- `scaDel`(k) removes the first $k \in \mathrm{NN}$ instructions of the program in a file; if the file contains a program of instruction length k or less it is replaced by $\#$.

calling an edit instruction reads: g`.scaPrefix`(a) prefixes the program contained in g with the atomic action a etc.

running a program from file. `scaCall` calls the operation of the program contained in a file. If the file is empty the program $\#$ is executed instead. The program (taken from the file) executes and its termination consists of handing control back to the calling program, which will continue with the first action following the call instruction (failing that an abort will result).

serialization. `scaAutoSerialize` takes the calling program and puts a text of that program object (including the action `scaAutoSerialize` being performed) into g (overwriting its current contents).

removing a file. `scaRmf` removes the file from the system. Further reference to it from the calling program will result in an immediate abort.

The following exercises exercises use these primitives:

1. Write a PGLA program which clears file g and edits the following program object $X = +b; !; (-c; !)^\omega$ into it.
2. Write a PGLA program which clears file g and edits the following program object $X = +b; !; (-c; !)^\omega; a; +c; !$ into it.
3. Write a PGLA program which declares file g and edits the following program object $X = a; (-c; !)^\omega; a; +c; !$ into it, then runs the edited program three consecutive times and thereafter removes the file and terminates.

[6] u can have the forms: $\#, \#k, a, +a, -a, !$.

5.3 System calls as program actions III

5.3.1 Impossibility of a batch-oriented halting information service

The next aim is to investigate the promise held by a batch-oriented halting information service, given the failure of the run-time approach to system-provided halting information.

Given the file management primitives of 5.2.1, it is reasonable to consider the following batch-oriented halting information service.

The system call `scaHib` can be applied to a file (calling g.`scaHib`). The system will return if the program contained in the file always terminates when set to run and `false` if it always fails. (In between both D and M may occur.) Always refers to the range of conditions under control of (or depending on) system users of any kind. Inspection of data that cannot be changed will be assumed to yield fixed outcome only. In particular: the system may use a fixed data base of halting information which it will never update.

The existence of a reliable batch-oriented halting service in a system taken for granted, the following complication arises: consider the PGLA program $X =$

$$\texttt{scaNewFile}g = \texttt{empty};$$

$$g.\texttt{scaAutoSerialize};$$

$$g.\texttt{scaPrefix}(\texttt{scaNewFile}g = \texttt{empty});$$

$$+g.\texttt{scaHib}; \#0; !$$

Consider the following questions:

1. Assume that X halts; verify under this condition that the call of g.`scaHib` will return true, making X diverge.

2. Assume that X diverges; verify under this condition that the call of g.`scaHib` will return false, making X halt.

3. Make an assessment of this conclusion: a well-defined distinction on halting versus non-halting, valid for programs sensitive to inquiries about halting only, does not exist. The concept is paradoxical, its being offered as a service not even even being an option.

5.4 System calls as program actions IV

5.4.1 Mobile software agents: departures

PGLA programs may be lifted to the status of mobile software agents in the presence of the file management primitives of 5.2.1. We assume the existence of locations '`site`k' for $k \in$ NN. Each site has its own system, able to run programs and offering the file management primitives mentioned above. In

addition a system call '`scaMail` g `to` k' is supported. The result of this action is that file g is mailed to `site` k where its contents will be run as a program.[7]

In order to add mobility to PGLA programming a new advanced control instruction is introduced:

$$\texttt{move\#\#}k$$

The effect of `move##`k is that the program following the instruction moves to `site` k, the running program performing no further actions. PGLAmsa extends PGLA with this move-instruction. A projection semantics for PGLAmsa can be found by using the file management primitives as follows. We provide an example, supposing that programs X, Y contain no moves (which is inessential): the projection `pglamsa2pgla` has this effect on the program X;`move##`k;Y. `pglamsa2pgla`(X;`move##`k;Y) =

$$X;\texttt{scaNewFile}\,g = \texttt{empty};$$
$$g.\texttt{scaAutoSerialize};$$
$$g.\texttt{scaDel(5)};$$
$$\texttt{scaMail}\,g\,\texttt{to}\,k;$$
$$g.\texttt{scaRmf};!;Y$$

Exercises:

1. Apply this projection with $X = a; b, Y = c; d$. Which file is mailed?
2. Define the working of the projection `pglamsa2pgla` in the general case.
3. Design a modification of the scheme supporting an action creating a remote clone of a running program.

5.5 Program algebra: alternative behavioral models for PGLA

5.5.1 Labeling actions with return values

We are interested in alteratives for the definition of the behavior for PGLA program X. In particular we intend to avoid the yield predicate, indenting to stay closer to standard process algebra. For every atomic action $a \in \Sigma$ we introduce two versions: $a_{\texttt{true}}$ and $a_{\texttt{false}}$. These actions represent a when being executed including the boolean that will be returned. The semantic equation for positive test actions then is this:

$$|+a;u;X| = a_{\texttt{true}} \cdot |u;X| + a_{\texttt{false}} \cdot |X|$$

Here $+$ expresses the alternative composition of two behaviors. Both possibilities are open; the environment will decide what happens (depending on the value returned).

[7] How the receiving site handles its 'arrivals' is the subject of exercise ??.

1. Give alternative equations for $|-|$ for all cases. It is helpful to assume that all actions invite a returned boolean. A behavior should be generated accepting a boolean even is it is not used.

2. Give modifications of the definitions of the use-oprator and the apply-operator.

5.5.2 Splitting actions in send and receive

A second modification is to introduce actions `r(true)` and `r(false)` representing the reception of a returned boolean from the environment. To obtain a uniform notation, the realization of an action a is also written as a send action (in the process algebra jargon): `s(`a`)` The following identity is plausible:

$$|+a;u;X| = \mathtt{s}(a) \cdot (\mathtt{r(true)} \cdot |u;X| + \mathtt{r(false)} \cdot |X|)$$

The following questions arise:

1. How to adapt all equations (in as far as needed).

2. How to adapt the definition of a coprogram and of the use-operator and the apply-operator, to mirror this modification. (Hint: coprograms need complementary send actions `s(true)` and `s(false)` as well as a complementary read action `r(`a`)` for all $a \in \Sigma$.)

3. In which cases can closed process expressions be found using alternative composition, sequential composition, binary Kleene star (iteration) and multi-exit iteration.

4. Prove that coprojection followed by projection from PGLA to PGLB and back is identity at the level of strong bisimulation semantics for this behavioral semantics definition. (Many similar questions arise. Proofs can be given in μCRL and can be proof-checked in `Coq`. Another option is to generate transition systems and to check for bisimulation using Aldebaran.)

5.5.3 Trace theory

Yet another option is to assign to a program X a pair $< \Sigma, V >$ with $\Sigma = \Sigma_l^{ps}(X)$ and V equal to the prefix closure of its set of execution traces (using return value labeled actions as above):

$$|+a;u;X| = \{a_{\mathtt{true}}\} \cdot |u;X| \cup \{a_{\mathtt{false}}\} \cdot |X| \cup \{a_{\mathtt{true}}, a_{\mathtt{false}}\}$$

Similar questions arise as in the previous semantic modifications.

5.6 Program algebra: timed action notation

1. We consider discrete time versions of these program algebras as follows: a jump instruction takes 0 times steps, independently of its counter size.

   (a) Consider the PGLA program $X = a; b; c; \#4; a; b; b; c; a; !$, assuming that each action takes 1 time step to execute (including '!'): how many time steps will the execution of X take?

   (b) Now assume that a takes 1 time step, b takes 4 time steps and c takes 7 time steps: how long will it take for X (as defined above) to terminate?

   (c) Consider the PGLA program $X' = a; b; +c; \#4; a; b; b; c; a; !$, and assume that all actions need 3 time steps. What is the duration of the two possible computations?

   (d) Consider the PGLA program $X'' = a; -b; +c; \#4; a; +b; -b; c; a; !$, and assume that all actions need 2 time steps. What is the duration of the longest possible computation?

2. Again we consider timed programs. Now the duration of the execution of an action is indicated as a superscript. The performing of jumps and termination takes no time.

   (a) How long does the execution of the PGLB program take?:

   $$a^5; b^2; \#3; \#8; a^{75}; c^0; d^1; d^1; \backslash\#5; !; !; !$$

   (b) How long do the two different possible executions of the PGLB program take:

   $$a^5; +b^2; \#4; \#8; a^{75}; c^0; d^1; d^1; \backslash\#5; !; !; !$$

   (c) How long does the longest possible execution of the PGLB program take

   $$a^5; +b^2; \#2; \#8; -a^{75}; +c^0; -d^1; +d^1; \backslash\#5; !; !; !; !$$

3. In addition to the superscripted actions with fixed duration there are actions with a variable duration. Such actions are not allowed to take more than some fixed time span. An indication of the maximal duration is given in a superscript carrying square brackets as follows:

   $$a^{[<750]}$$

   is an action which performs like a for 750 time units. If a has been completed within that time its boolean is returned and the next instruction is processed. If a has not terminated within 750 time units its execution is interrupted in the 750th time slot. The returned boolean is `false`. A boolean `true` can be returned until time slot nr. 749.

(a) How many time units are needed at most to execute the following PGLD programs:

$$c^{20}; b^{[<200]}; e^{[<100]}; d^{25}$$

$$-c^{20}; +e^{[<1000]}; \#\#10; b^{[<200]}; e^{[<100]}; d^{25}$$

(b) The action a, used without any superscript, may take any number of time units. Is there a finite upperbound for the execution times of the following PGLD programs:

$$\#\#4; d^{25}; e; f^{[<5000]}$$

$$\#\#3; d^{25}; -e; f^{[<1000]}$$

4. By adding a superscript ndg to a jump instruction it is indicated that executing the jump will take as many time units as the jump counter has digits (in decimal representation counting meaningful digits only). If no superscript is provided the time required for the jump is 0. In PGLA and PGLD the same superscript can be used to indicate the time required for a termination instruction.

(a) With these conventions determine the longest execution of the PGLD programs:

$$-c^{20}; +e^{[<1000]}; \#\#^{ndg}10; b^{[<200]}; e^{[<100]}; d^{25}$$

$$-c^{2}; +e^{[<100]}; \#\#^{ndg}10; b^{[<200]}; \#\#256; e^{[<100]}; d^{25}$$

(b) Does the following PGLA program have a terminating execution taking between 1000 and 2000 time units:

$$a^{10}; +b^{[<35]}; (\#^{ndg}02; c; -d^{[<350]}; !^{2500}; +b^{0}; !^{250}; c^{1})^{\omega}$$

5.7 Program algebra: sequential composition

For two programs X and Y the sequential composition $X \ominus Y$ works as follows: X is performed; if it arrives at a termination action, that action is skipped and the execution of Y is started. Upon termination of Y the entire $X \ominus Y$ terminates. In some cases it is not easy to express $\ominus$ in a certain program notation. For instance in PGLA an expression for $X \ominus Y$ results from a rather involved transformation to be applied to X and Y. In the following cases X and Y are in the same notation, and the exercise is to express $X \ominus Y$ in that notation as well.

1. In PGLA

(a) $X = a$, $Y = b$

(b) $X = a; !$, $Y = b; !$

(c) $X = -a; !; c; !$, $Y = b; c; !$

(d) $X = a;(+b;!;c)^{\omega}$, $Y = d;!$

2. In PGLB $X = c;+d;\#4;e;e;f$ and $Y = g;+h;\backslash 2;f$

3. In PGLC $X = a;+b;\backslash\#2;\#3;c;c$ and $Y = c;-d;\backslash\#2;f$

4. In PGLD $X = a;b;c;+d;\#\$8;e;f;g$ and $Y = e;e;+e;\#\#7;f;f;g;g$

5. Provide general procedures to find the sequential composition of two programs for PGLA, PGLB, PGLC, PGLD. (Preferably avoid translating both programs to PGLA first, a solution that is correct but 'expensive', failing to exploit the special properties of certain notations.)

5.8 PGLDci and PGLDcit

1. Describe a co-projection from PGLDci to PGLDg (assuming that programs to be co-projected are ICCC).

2. Design a projection from PGLDg to (ICCC programs in) PGLDci, such that from composition with the projection from PGLDci to PGLD an adequate projection from PGLDg to PGLD results. (In other word factorize the projection from PGLDg to PGLD through PGLDci.)

5.9 Coprograms

Let S be a state space with effect functions and yield functions for all actions in Σ. `coprog`(Σ) denotes the set of all coprograms with upper signature equal to Σ.

1. Determine a natural mapping `Cpr(-)` from S to `coprog`(Σ).[8] Verify:

$$effect_a(s) = (\texttt{Cpr}(s))_{\partial a}$$

2. Let p and q represent different points in the state space S. perform a name refinement to the actions ocurring in Σ by prefixing each name with T or F.

3. Explain how the state space with two groups of refined actions (one for p and one for q) corresponds to

$$p.\texttt{Cpr}(X) \oplus q.\texttt{Cpr}(X)$$

A significant difference between the state space and coprograms is obscured by the following questions: whereas the coprogram is modified by a program only, the state can be changed during its interaction with a program. Each state of the state space can be considered a mere observation post in a larger world. The observations may change independently from any programmed influence.

[8] Hint: `Cpr`(s) describes the behavior of S with s taken as the point of departure.

5.10 Coprogram calculus

1. Prove the following properties of emulation equivalence:

$$H_1 \leq_{emd} H_1 \oplus H_2 \text{ if } \Sigma_u(H_1) \cap \Sigma_u(H_2) = \emptyset$$

$$\mathtt{f}.H \equiv_{emd} H$$

$$H_1 \oplus H_2 \equiv_{emd} H_2 \oplus H_1$$

$$(H_1 \oplus H_2) \oplus H_3 \equiv_{emd} H_1 \oplus (H_2 \oplus H_3) \text{ if } \Sigma_u(H_1) \cap \Sigma_u(H_2) \cap \Sigma_u(H_3) = \emptyset$$

   Which of these conditional equivalences hold for coprogram calculus. Give counter examples to the validity of the other equivalences in coprogram calculus. (In other words: what goes wrong if $\equiv_{emd}$ is replaced by $=$?)

2. Explain why the conditions on empty interface intersections are needed. Provide counter examples in case the conditions are left out.

5.11 Recursion, PGLDg$_\mu$, PGLDgr

1. Design a projection `pglsgm2pgldg`, making only one copy of the series of test instructions needed to realize a 'dynamic goto' in PGLDg

2. Let PGLDgfr be a program notation with so-called flat recursion: nowhere in the execution is the recurion depth above 1 (after every returning jump is a return instruction). Design a projection from PGLDgfr into PGLDgμ (using a natural number register rather than a stack).

5.12 Multi-threading, PGLDgmt

1. Design a modification PGLDgmt/rot of PGLDgmt, allowing `rotate` as an advanced control action which must be used in order to change the order of thread items. Design a simplified projection from PGLDgmt/rot to PGLDgμ. Then design a projection from PGLDgmt to PGLDgmt/rot, thus establishing PGLDgmt/rot as a reasonable intermediate language for the projection of PGLDgmt to PGLDgμ.

2. Design a version PGLDgmt/rot/ind of PGLDgmt/rot where fork actions need a sequence number as an extra parameter. This parameter indicates where in the thread vector the new item will have to be placed. Provide a direct projection semantics for PGLDgmt/rot/ind by projecting it onto PGLDgμ. Demonstrate that PGLDgmt/rot/ind can be used as an intermediate notation for the projection from PGLDgmt/rot to PGLDgμ.

5.13 multi-threading, PGLDgmt

Write a PGLDmt program which accumulates an unbounded number of threads during execution.

6 Project level

6.1 Program algebra: PGLA

1. We recall the file management primitives of paragraph 5.2.1. The system call `scaAutoSerialize` can be viewed as an advanced control instruction as well. This requires a significantly more involved projection, the effect of auto-serialization now being generated by an appropriate sequence of edit actions, these edit actions in turn being generated by the projection operator. Investigate the details of this matter.

2. The appendix of 'Program algebra for component code' contains syntax and semantic equations for the compound instruction operator. The compound $\{X\}$ wraps X in an instruction of length 1, being relevant for jumps flying over it. PGLAcmp denotes the extension of PGLA with compound instructions, its projection semantics still missing. Design a projection semantics for PGLAcmp. It is practical to use a coprojection from PGLAcmp to PGLDg first.

6.2 Dataflow and PGLDg, PGLDgc

1. Develop an extension PGLDc of PGLD with a conditional construct, using almost the same syntax for the conditional control actions as in PGLDgc.

2. Define a projection `pgldc2pgldgc`, thereby defining the semantics of PGLDc via PGLDg.[9] Explain why it is harder to directly design a projection from PGLDc to PGLD.

6.3 Recursion, PGLDgr

1. Design a syntax for the declaration and the calling of (recursive) procedures. For instance '`callProc` k' can be used to express the calling of procedure k. Define a language PGLDgp extending PGLDg with your primitives. Find a projection from PGLDgp to PGLDgr.

2. Investigate how to deal with the option that all procedures have an NN parameter (called by value).

6.4 Multi-threading, PGLDgmt

1. Consider the following PGLBgmt programs. How many threads[10] can be active simultaneously (maximum during computation):[11]

[9] Hint: first introduce label catches in front of every action.

[10] The number of threads during a stage in the computation is measured as the length of the thread vector.

[11] Finding reliable answers to these questions seems to be quite hard. For the more complex cases making a computer simulation of the execution model may be the most practical solution.

(a) $X_1 = a; \mathtt{fT\#\#}\pounds 3; c; \pounds 3; d$

(b) $X_2 = a; \mathtt{fT\#\#}\pounds 3; b; \mathtt{fT\#\#}\pounds 3; c; \pounds 3; d$

(c) $X_3 = -f; \mathtt{fT\#\#}\pounds 3; +f; \mathtt{fT\#\#}\pounds 3; c; \pounds 3; d$

(d) $X_4 = a; \mathtt{fT\#\#}\pounds 3; b; \mathtt{fT\#\#}\pounds 3; c; \pounds 3; d; \mathtt{fT\#\#}\pounds 3; e;$

(e) $X_5 = X_1; X_2; X_3; X_4$ fTendenumerate

(f) Consider the coprogram cptv. When a particular program is being translated from PGLDgmt to PGLDg only a limited number of current labels and of priorities will play a role. This suggests the following steps (constituting exercises as well):

i. Design and specify a coprogram $\mathtt{cptv}_k$ which is like cptv but deals with priorities and label values below $k \in$ NN only. Verify that $\mathtt{cptv}_k$ is a data structure coprogram.

ii. Design and specify an attractive memory compogram cpdam for a direct access memory indexed by NN. 'Attractive' refers to the possibility of using cpdam for programming as well as to having it realized in hardware directly.

iii. Design a program aggregate constructing $\mathtt{cptv}_k$ from a number of disambiguated copies of cpdam.

iv. Take X in PGLDgmt such that its translation will involve no more than k labels and priority levels. Combine the results of this exercise with the projection known for PGLDgmt in order to find a projection using the disambiguated copies of cpdam instead of $\mathtt{cptv}_k$.

The value of the projection found in this exercise is in using coprograms directly available in hardware.

(g) Extend PGLDgmt to a program notation PGLDgmtn in such a way that each thread upon creation is assigned a natural number as its name. More than one thread may carry the same name at some instant of time.

i. Suggest a modification of the syntax for the fork instruction.

ii. Secondly add a primitive for the 'remove instruction' allowing one thread to stop the existence of another thread (identified by its name). If there is more than one item carrying the name mentioned in the 'remove' instruction, the left-most item in the thread vector is removed.

iii. Explain the semantics of these constructs in natural language (are there any alternative options to be taken into acount?).

iv. Modify the coprogram thread vector in such a way that the new language can be given a projection semantics.

v. Give a projection semantics for this new language.

(h) Combine multi-threading and recursion. This requires a significant extension of the thread vector as every thread item must now carry its own stack along with it. Several new service interface actions for the thread vector are to be designed and specified and the projection must be reconsidered and extended with clauses for $\rho\#\#\pounds k$ and ρ.

6.5 Co-ordination techniques for multi-threading.

We consider the following new advanced control instruction:

$$\rho\,\mathtt{meth}\,k\#\#\pounds l$$

The intended meaning being as follows: the instruction corresponds to a returning jump (just as $\rho\#\#k$), having the additional property of coloring the thread with a method name ($k \in$ NN). The method name is also stored on the stack; upon execution of the next return instruction, the thread will return to the previous method. Initially threads are in a default method -1. Being in a method can have several concrete interpretations: for instance in OO-languages it corresponds to having made a jump to the body of method (of an object or a class) and being in the process of executing that code. More down to earth it can mean that the program counter points to a memory segment (storing the code for a compiled method body e.g.).

If full OO-execution is to be modeled, object instance identities and class names are to be tagged onto the method name, thus obtaining a more refined bookkeeping. These additional aspects are of no particular relevance to the foremost connection between method coloring and multi-threading: method coloring induces a run-time equivalence relation on thread items, thread items being called equivalent simply if they have the same current method (color). Several co-ordination techniques have been developed. The projects consist of modeling these additional features by finding new instructions, extensions of the thread vector structure, as well as enhanced projection operators. We describe two thread co-ordination techniques.

1. Explicit wait and release.
   By three instructions, threads can be told to wait or can be released from waiting: `Wait()`, `notify()` and `notifyAll()`. If a thread with method color k executes `wait()` it starts waiting until its turn has come to accept a `notification` message. Notification messages are generated if another thread, having the same color k, excutes an instruction `notify()`. The waiting threads are put in a queue and an action `notify()` enables just one of the waiting threads (if any) to proceed. If no waiting threads are on the lookout for a helpful `notify()`, the message gets lost. A `notifyAll()` instruction is like the `notify()` but with the power to unblock all waiting threads (of the same color of course) at once.

2. Synchronized methods.
   An additional refinement of the method colored returning jump is as fol-

lows:

$$\rho\,\texttt{syncMeth}\,k\#\#\pounds l$$

The additional effect of the indication that the method color is 'sync' (synchronized) is to regard the instruction as a move into a queue of threads waiting for admission to method k. Only if all other (sync) threads have left method k, another waiting thread can be admitted. Threads not using the sync attribute when jumping to another method color may enter at once. Waiting threads wait only to see other sync threads leave the relevant color.

6.6 System calls as program actions V

6.6.1 Mobile software agents: arrivals

This exercise is needed to complete the picture regarding mobile sotfware from previous exercises. It was mentioned in 5.4 that mobile agents are wrapped into mailed messages, travel to another site and appear in the mailbox of the local site manager at arrival. The site manager will open the mail and make the mailed programs come to expression. It will be sketched how that works, the exercise being to elaborate on the details. The local site manager is just a milti-threaded program, splitting off a new thread for each opened mail. It is in a permanent loop, alert to its mailbox, reading a message whenever it is present. Its code looks as follows:

$$\pounds 0; +\texttt{scaMailBoxEmpty}(); \#\#\pounds 0;$$

$$\texttt{scaNewFile}\,g = \texttt{scaMailBox.get}();$$

$$\texttt{fT}\#\#\pounds 0;$$

$$g.\texttt{scaStartCall}();$$

$$\pounds 1; g.\texttt{scaStepCal}();$$

$$-g.\texttt{scaEndCall}(); \#\#\pounds 1;$$

$$g.\texttt{scaRmf}$$

Five system calls make their first appearance in this program:

start interpreter. g.`scaStartCall()` initiates the expression of the contents of file g as a running program, preparing to do this in a step-by-step fashion, each following step requiring a call of g.`scaStepCall()`.

continue interpretation. g.`scaStepCal()` makes the run of the contents of g progress with one step.

decide to proceed. g.`scaEndCall()` decides whether the run has come to an end.

get mail from mailbox. `scaMailBox.get()` takes the oldest entry from the mailbox. It can be stored in a file, for instance the newly defined g in the program, thus resulting in the system call `scaNewFile`g = `scaMailBox.get()`.

check for new mail. `scaMailBoxEmpty()` is a test, succeeding if new mail has arrived.

The design of the first three of these instructions is rather involved. It requires decoding of mails, thereby depending on the level of abstraction of message transmission.