University of Amsterdam
Programming Research Group

Projection Semantics for Multi-file Programs

J.A. Bergstra
P. Walters

Report PRG0301 May 2003

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

P. Walters

Microsoft
the Netherlands

e-mail: pwalters@microsoft.nl

Projection semantics for multi-file programs

Jan Bergstra*
University of Amsterdam
Utrecht University

Pum Walters†
Microsoft

May 8, 2003

Abstract

The multi-file paradigm – where program modules are located in different files – as exhibited in Java, is investigated using the program algebra PGA. In order to so a number of auxiliary results in the context of PGA are presented: languages with explicit location of execution (PC), method invocation, structured programming, and a flat file system.

1 Introduction

In a desire to understand and reason about the nature and mechanism of programming, the program algebra project has developed the program algebra PGA ([BL02]). PGA offers an algebraic theory intended to facilitate reasoning about sequential programs. By definition, a sequence of instructions is a program, as is any entity the meaning of which can be defined by mapping it to a sequence of instructions. Such a map is called a projection. In this setting, aspects of programming are discussed, based upon definitions that are open to debate and formal reasoning.

The programming language Java refines a programming paradigm that exists in various languages in more or less ad-hoc fashions and that we will call *multi-file programs*: program modules originate from multiple separate files, and the content of such a file refers to (calls) methods contained in another file.

In this article, our purpose is to define a very simple format for multi-file programs in the style of PGA. Then, we will identify a fragment of Java that we can study using this format.

In order to achieve this, a number of prerequisites have to be developed:

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

†H.R. Walters, Microsoft, The Netherlands, email: pwalters@microsoft.com

Explicit PC.
Control features such as a method call require the ability to represent the program counter (i.e., the locus of execution) explicitly, to store, retrieve and manipulate it.

In computer hardware, the program counter is a special register which holds the address of the current instruction. Maintaining the proper value at all times, which includes an increment each time an instruction is fetched and performed, is ordinarily done automatically by hardware and does not require explicit program control. In addition, advanced control instructions manipulate the contents of this register as if it is a data register, storing it, altering it or replacing it with a stored, or entirely new value.

The "locus of execution" in PGA is implicit; thinking of it as data leads to defeat of the purposes of PGA. As it turns out though, the mechanisms at our disposal to use or manipulate the locus are sufficient for most control features.

Our intention is to use a value that can be manipulated and that can be transfered to and from the PGA's implicit locus of execution; we do not alter the nature of the locus of execution, or the fact that it remains implicit, and we do not expect this value to be synchronous with the locus of execution with every instruction: only after specific control instructions. One can think of this value as the hardware PC at the moments it is being manipulated as data. We shall refer to it as the PC.

Invocation and returning goto.
Since we intend to look at Java, where method invocation is the predominant control flow mechanism, we require a PGA-language which has that feature.

In [BB02] the feature is defined using a projection based on primitives which are not present in our current framework (notably: molecular programming primitives). Accordingly we redefine this feature.

Vector labels.
PGA and various related languages use natural numbers as labels. Although this is quite adequate in theory, in practice many formulations become cumbersome in order to avoid unintended clashes of labels. Vector labels provide a clearer approach.

Flat file system.
In order to reason about multi-file programs we develop a (minimal) formalization of a file system.

Structured programming.
In structured programming a program is regarded as a hierarchy of nested blocks. A block may be executed or may be passed over entirely.

1.1 PGA

PGA is an algebra for sequential programs, focusing on what is traditionally called "control flow". PGA abstracts from data, assuming all data manipulation to be managed by the set of *basic instructions* Σ. The sole link between data and control flow is embodied in the assumption that every basic instruction returns a boolean value which may or may not be examined by the program.

PGA also abstracts from most syntactical aspects: PGA values, i.e., base programs, are non-empty, possibly infinite, sequences of the so-called *primitive instructions* shown below (here, a ranges over Σ).

- a: perform a, and disregard the boolean value that is returned;
- $+a$: perform a, and, if the returned boolean is *true*, continue with the next instruction. Otherwise, skip the next instruction and continue with the subsequent instruction;
- $-a$: perform a and continue with the next instruction if the returned boolean was *false*, skipping one instruction otherwise.
- **!**: terminate execution;
- $\#n$: continue execution at the nth subsequent instruction "goto" (there is no backward jump in base programs; loops are unrolled where necessary).

PGA terms are constructed from primitive instructions, from the associative operator $-;-$ for sequential execution, and the operator $-^\omega$ for infinite repetition. Four equations (schemes) are given for PGA terms: $(X;Y);Z = X;(Y;Z)$ (associativity of ;), $(X^n)^\omega = X^\omega$ (for any positive natural n), $X^\omega;Y = X^\omega$, and $(X;Y)^\omega = X;(Y;X)^\omega$. Terms equal with respect to these equations are *instruction sequence congruent* ($\equiv_{isc}$). Any finite closed PGA term has a $\equiv_{isc}$ canonical form X or $X;Y^\omega$ (where X and Y are finite primitive instruction sequences).

The *behavior* of programs is defined as follows (here, P and Q range over program behaviors, and a ranges over Σ).

- $P \trianglelefteq a \trianglerighteq Q$: the program performs a. If the boolean returned is *true* the program proceeds with behavior P, and otherwise with behavior Q. Note that all possible behaviors of a program are considered rather than the behavior during one particular execution;
- S: the program terminates;
- D: the program is inactive and will not process further instructions. For example, it performs an infinite empty loop;
- The sequentialization notation $a \circ P$ is introduced as a short hand for $P \trianglelefteq a \trianglerighteq P$.

The behavior extraction operator |-| assigns behavior to PGA-programs in the straightforward manner. Programs are behavior equivalent ($\equiv_{be}$) if they have the same behavior.

A *projection* is a map ϕ from a language L (set of objects) into the set of base programs for some appropriate Σ. Such a map defines a meaning for L-programs as the behavior of their ϕ-value. We write $|X|_L = |\phi(X)|$ for $X \in L$.

Finally, [BL02] defines several languages, some of which are discussed below.

1.1.1 PGA-related languages introduced in [BL02]

The language PGLA offers a textual representation of PGA. The representation of $-^\omega$ is the instruction $\backslash\backslash\#k$. This instruction has no equivalent in any programming language known to us). In addition it has non-local implications which make it difficult to handle consistently.

In PGLB this situation is improved upon by removing that instruction and introducing a more conventional backward (relative) jump.

PGLD offers an absolute jump ($\#\#n$) as its predominant control mechanism. Although unfriendly for a human programmer, this mechanism is a suitable target in projections.

PGLDg removes the detailed technicalities of absolute addressing; it offers (numeric) labels ($£k$) and a jump-to-label ($\#\#£k$) instruction.

PGLE is identical to PGLDg, except that it requires conditional basic instructions only to be followed immediately by a jump instruction or a terminate instruction. This avoids the 'true-case' falling through into the 'false-case', which is unclear and leads to all sorts of technicalities in program transformations.

[BL02] introduces other languages, but they shall not be used in this article.

1.1.2 Co-programs and Focus Method Notation

Since a basic instruction results in a boolean, it follows that something outside the program produces that boolean. Whether this is trivial, perhaps producing the same boolean for the same instruction each time, or more complex, perhaps producing the binary digits of an approximation of one solution of a fifth-degree polynomial equation, is left open in PGA. Indeed, PGA leaves entirely open how basic instructions come to their result.

An entity performing services for a program is called an *instruction executing agent*, a *co-program*, or a *re-actor*. One class of co-programs, called *state machines*, will be introduced in the next section.

The *focus method notation* (FMN) is a convention which addresses basic instructions. It distinguishes two aspects in a basic instruction: the *focus*, which one can think of as the name a program uses to identify one particular co-program[1], and the *method*, which denotes the service required of that co-

[1]Note that a focus doesn't identify a specific co-program; merely the name under which a program addresses one of possibly multiple co-programs. This is similar to many programming languages, where programs identify files using handles.

program, and possibly additional information needed to perform the service. In FMN, focus and method are separated by ".". Both focus and method must start with a letter and may contain only alphanumeric ASCII characters and the colon symbol ":".

1.2 State Machines

A state machine ([BP02]) is a pair $\langle \Sigma, F \rangle$ where Σ (the interface of the machine) is a set of basic instructions, and F is a map from sequences of such instructions (Σ^+) to the booleans, where $F(a_1; \ldots; a_n)$ signifies the boolean value resulting from execution of a_n in the state resulting from the sequence of instructions $a_1; \ldots; a_{n-1}$.

For example, `smbv`, a state machine for a boolean variable, arguably the simplest abstract data type, is $\langle \{\texttt{set:true}, \texttt{set:false}, \texttt{eq:true}, \texttt{eq:false}\}, F_{bv} \rangle$ where $F_{bv}(a_1; \ldots; a_{n-1}; a_n)$ equals *true* if $a_n \equiv \texttt{eq:true}$ and the last 'set' instruction in $a_1; \ldots; a_{n-1}$ is `set:true`, or $a_n \equiv \texttt{eq:false}$ and $a_1; \ldots; a_{n-1}$ contains no 'set' instruction or the last one is `set:false`, and $F_{bv}(a_1; \ldots; a_{n-1}; a_n)$ equals *false* otherwise. Observe that $F() = \mathit{false}$. By convention, this is the case for every state machine.

Different state machines can be combined, when their interfaces are disjoint. The combined interface is the union of the separate interfaces, and the combined map is the obvious extension of the two separate maps. A state machine calculus is defined in [BP02].

The interaction between program behaviors and state machines is embodied in two binary operators: the *use operator* ($/_f$) and the *apply operator* ($\bullet_f$), which have a program behavior and a state machine as arguments, and where f is a focus occurring in the program. The set of methods occurring in the program for that focus must be contained in the interface of the state machine.

The apply operator considers a program (behavior) as a transformer of state machines in that basic instructions alter the state of the state machine. The use operator considers a state machine as a transformer of behaviors in that the state machine simplifies the program by offering a decisive choice on the conditional instructions with the given focus.

As an illustration we offer one of the defining equations of the use operator:

$$(P \trianglelefteq f.m \trianglerighteq Q) \mathbin{/_f} H = (P \mathbin{/_f} \tfrac{\partial}{\partial m} H) \triangleleft F(m) \triangleright (Q \mathbin{/_f} \tfrac{\partial}{\partial m} H)$$

Here, $H = \langle \Sigma, F \rangle$, and $\frac{\partial}{\partial a} H$ signifies the state machine obtained from H assuming a to have been the first instruction, and $.\triangleleft.\triangleright.$ signifies a three-valued conditional.

2 Projection-compatibility

The purpose of PGA is to make explicit and then study programs, and to reason about them formally, by mathematical or automatical means. Adding arbitrar-

ily complex co-programs, or co-programs which are beyond formal reasoning altogether, defeats this purpose.

Even when a co-program does allow for formal reasoning, there is a price attached to its use: results which depend on specific properties of that co-program have limited usefulness in a more general scope. General results remain limited to programs where one can abstract from the behavior of co-programs in a well understood manner.

Because of this we introduce the notion of projection-compatibility. A co-program is *projection compatible* on a class of programs, if a projection exists from programs in that class that use the co-program to programs with the same behavior, that do not use that co-program (or any replacement).

If the domain of such a projection is an entire language (such as PGLD), the class of programs is implicitly limited to that language and to any further languages for which the meaning has been defined by a map to that language. Accordingly, a compatibility projection on PGA (or PGLA) applies to all languages in the PGA framework. From a practical perspective we mention that PGA and PGLA can be embedded, preserving meaning, in PGLB, PGLD, and certain other languages. Accordingly those languages can also be taken as a point of departure to establish general projection compatibility.

If that domain is characterized by other properties (whether within one language or not), the applicability to further languages depends on what properties on such languages lead to the desired properties in their meaning.

Focusing on state machines, we mention three facts:

- State machines with finitely many states are projection compatible on all programs. This is a special case of the third fact.

  As an example, consider the state machine $\langle\{\mathtt{prime}{:}i|i \in I\!N\}, F\rangle$, where $F(\ldots;\mathtt{prime}{:}i) = \mathit{true}$ precisely when i is prime. This state machine has a single state, and the projection unpr, defined below, maps PGLA programs using this state machine on focus $\mathtt{pr}$ to behaviorally equivalent programs not having this focus.

  $$\mathit{unpr}(u_1;\ldots;u_n) = \phi(u_1);\ldots;\phi(u_n)$$

  Here,

  - $\phi(\mathtt{pr}.m) = \#1$ for any method m;
  - $\phi(+\mathtt{pr.prime}{:}p) = \#1$ when p is prime;
  - $\phi(+\mathtt{pr.m}) = \#2$ in any other situation;
  - $\phi(-\mathtt{pr.prime}{:}p) = \#2$ when p is prime;
  - $\phi(-\mathtt{pr.m}) = \#1$ in any other situation;
  - $\phi(u) = u$, otherwise;

- A priori, state machines with infinitely many states are not projection compatible. This follows from the fact that a PGA program without co-programs has finitely many states.

- A state machine is projection compatible on a class of programs if each program in that class can only make the state machine reach finitely many states.

  In this article we will not prove this fact in general (we will look at some instances), merely stating it as a conjecture. That it is reasonable can be seen for example by considering the following projection ϕ on PGLD. Let p be a program which uses state machine s via focus f, and which potentially makes s reach the states $\{s_1, \ldots, s_n\}$. Then, $\phi(p)$ contains precisely n copies of p which have been transformed as follows:

  - A basic instruction (void or conditional) at location l in p with focus f which changes state s_i to state $s_{m_{s,p,l}(i)}$ is replaced in copy i by the goto instruction to location $l+1$ in copy $m_{s,p,l}(i)$;
  - If the basic instruction at location l was a positive condition and s in state s_i results in false, for the basic instruction at location l in p, or it is a negative condition and that result is true, then the goto instruction produced above is replaced by the goto instruction to location $l+2$ in copy $m_{s,p,l}(i)$;

  In other words: state changes are replaced by jumps to the appropriate copy, and resulting booleans reflect the copy they are contained in.

  As an extended example the next section (2.1) discusses projection compatibility along these lines for a state machine to be used elsewhere in this article.

Establishing behavior equivalence between programs using a projection compatible co-program and their image under a projection compatibility map is often difficult to prove in general, even in straightforward cases such as our "prime" example, above. However, it is often sufficiently satisfactory to establish bisimulation for a number of representative programs.

In this document we will offer projections wherever projection-compatibility is claimed, but we will neither prove projection-compatibility in general, nor establish it for specific programs.

Finally we stress that the purpose of projection compatibility is not to avoid the use of co-programs; they are an integral part of the PGA framework. Also, the compatibility projection of a program is not to be confused with the meaning of that program, which may very well include co-programs.

In the remainder of this article, we use the notation $l::f\text{-}m$ for programs written in language l using a focus f with methods defined in interface Σ_m. An $l::f\text{-}m$ program uses advanced control instructions defined in l, but assumes the presence of a co-program which offers services defined in Σ_m and which is used by the program via focus f.

2.1 The state machine `smpc`

State machine `smpc` is used to store a number. It has instructions `set:`n, which set the current value to n, forgetting any previous stored value, and

eq:n which return true precisely when the current value equals n. That is, smpc $= \langle \Sigma_{pc}, F_{pc} \rangle$, where $\Sigma_{pc} = \cup_{i \in \mathbb{N}} \{\texttt{set:}i, \texttt{eq:}i\}$, and $F_{pc}(s) = \mathit{true}$ when $s \equiv b_1; \ldots; b_n; \texttt{set:}k; c_1; \ldots; c_m; \texttt{eq:}k$, for $n, m \geq 0$, where each $b_i \in \Sigma_{pc}$ and each $c_i \equiv eq : l_i$ for some l_i, and $F_{pc}(s) = \mathit{false}$ in any other situation.

State machine smpc is projection compatible on PGLD, and without loss of generality we consider PGLD programs using methods of that state machine via focus pc (i.e., PGLD::pc-pc). We take PGLD as our point of departure because it leads to a more straightforward projection than would PGLA or other languages. As mentioned, this is sufficient to establish projection compatibility on all PGA languages.

Although smpc has infinitely many states, only finitely many states can occur, since the state of smpc is determined exclusively by the final set instruction (or absence thereof), and no program contains more than a finite number of such instructions. This is the basis of our projection from PGLD::pc-pc to PGLD.

The projection copies the program as many times as there are distinct states that can be reached or queried. Then, the fragment processing occurs in encodes the current state; processing pc.set:k corresponds to branching to the corresponding fragment; and querying the current state with pc.eq:k succeeds depending on the actual fragment.

The projection from PGLD::pc-pc to PGLD of a program $p = u_1; \ldots; u_m$ with focus pc is defined as follows. Let $\widehat{L} = \{l_2, \ldots, l_k\}$ be the ordered set of values occurring in p in a basic instruction with focus pc and method set or eq, ($\widehat{L} = \{l | \exists i : u_i \equiv \texttt{pc.set:}l \vee u_i \equiv \texttt{pc.eq:}l\}$), and let $L = \{l_1, \ldots, l_k\} = \widehat{L} \cup \{l_1\}$ where l_1 is some arbitrary value not occurring in $\widehat{L}$ (this value will be used to represent the initial state of the state machine, where the 'stored' value, by definition, is not equal to any specific value). Then,

$$\mathit{pgld}\text{::}\mathit{pc}\text{-}\mathit{pc2pgld}(p) = \phi_k^{m,L}(p)$$

Here, ϕ and auxiliary functions are defined as follows. Notation $l^m(i,j)$ signifies the location of the j-th instruction in fragment i.

- $\phi_1^{m,L}(u_1; \ldots; u_m) = \psi_{1,1}^{m,L}(u_1); \ldots; \psi_{1,m}^{m,L}(u_m)$;
- $\phi_{i+1}^{m,L}(u_1; \ldots; u_m) = \phi_i^{m,L}(u_1; \ldots; u_m); \#\#0; \#\#0; \psi_{i+1,1}^{m,L}(u_1); \ldots; \psi_{i+1,m}^{m,L}(u_m)$;
- $\psi_{i,j}^{m,L}(\#\#l) = \#\#t$ where $t = l^m(i,l)$ for $1 \leq l \leq m$, and $t = 0$, otherwise;
- $\psi_{i,j}^{m,L}(\texttt{pc.set:}l_n) = \#\#l^m(n,j)$+1;
- $\psi_{i,j}^{m,L}(+\texttt{pc.set:}l_n) = \#\#l^m(n,j)$+2;
- $\psi_{i,j}^{m,L}(-\texttt{pc.set:}l_n) = \#\#l^m(n,j)$+1;
- $\psi_{i,j}^{m,L}(+\texttt{pc.eq:}l_n) = \#\#l^m(i,j)$+1, when $i = n$;
- $\psi_{i,j}^{m,L}(+\texttt{pc.eq:}l_n) = \#\#l^m(i,j)$+2, otherwise;

- $\psi_{i,j}^{m,L}(-\texttt{pc.eq:}l_n) = \#\#l^m(i,j)+1$, when $i \neq n$;
- $\psi_{i,j}^{m,L}(-\texttt{pc.eq:}l_n) = \#\#l^m(i,j)+2$, otherwise;
- $\psi_{i,j}^{m,L}(\texttt{pc.eq:}l_n) = \#\#l^m(i,j)+1$;
- $\psi_{i,j}^{m,L}(u) = u$, otherwise;
- $l^m(i,j) = (i-1)*(m+2)+j$.

Note that $\psi_{i,j}^{m,L}$ is somewhat cumbersome in order to handle 'rogue' programs that contain instructions such as $-$`pc.set:`l or `pc.eq:`l. Though not forbidden, these instruction are odd in the context of `smpc` in that `set:`l always yields *false*, and `eq:`l has no effect in a void basic instruction.

We conjecture that the behavior of a PGLD::pc-pc program using `smpc` via focus `pc` is identical to that of the projected program. That is, given a PGLD::pc-pc program p, $|p|_{\text{PGLD::pc-pc}} \,/_f\, \texttt{smpc} \equiv_{be} |\mathit{pgld}\text{::}\mathit{pc\text{-}pc2pgld}(p)|_{\text{PGLD}}$.

As mentioned, this establishes projection compatibility on all PGA languages.

The following examples may serve to illustrate the projection.

$$\mathit{pgld}\text{::}\mathit{pc\text{-}pc2pgld}(\texttt{pc.set:}1; a; +\texttt{pc.eq:}1; b; -\texttt{pc.eq:}1; c) =$$
$$\#\#10; a; \#\#5; b; \#\#6; c; \#\#0; \#\#0; \#\#10; a; \#\#12; b; \#\#15; c$$

3 Relocatable programming with explicit PC

PGLB is the PGA language for relocatable programming. *Relocatable programming*, also referred to as program counter relative programming, means that addresses in an instruction are given relative to the location of that instruction.

In many compilers relocatable code generation is an option; it is selected when compiling libraries, for instance, leading to library modules which can be added to programs at an arbitrary location, without an expensive relocation phase. In ordinary code the option is not selected, because absolute addressing is generally faster.

In the context of PGA this property is relevant because it simplifies projections significantly: under mild restrictions fragments can be concatenated without alterations, and the behavior of the resulting program can be understood from the behavior of the fragments. Without formalizing this, we shall refer to this property as *monotony*.

PGLBpc is an extension of PGLB in which the locus of execution can be stored and retrieved. In addition to PGLB, PGLBpc has the following instructions. We refer to the stored value as PC.

- `pc=here`
  Set PC to the location of this instruction. One can think of this instruction as assigning PGA's locus of execution to PC;

- `pc=here+`n
  Set PC to the location of the n-th instruction ahead, if that exists, or terminate, otherwise;

- `pc=here-`n
  Set PC to the location of the n-th previous instruction, if that exists, or terminate, otherwise;

- `##pc`
  Jump to the instruction the location of which is held in PC, or terminate if no such instruction exists. One can think of this instruction as assigning PC to PGA's locus of execution.

As an example, consider the following template, where p is an arbitrary PGLB program (we are assuming p proceeds to execute it first subsequent instruction when it has finished, as is reasonable in a sequential programming environment):

`pc=here+1;` p `; +a; ##pc`

Fragment p is repeated while basic instruction `a` succeeds (i.e., this is a template which implements "`do` { p } `while(a);`" in Java syntax). Note that the size of p doesn't matter; it is impossible to express such a template in PGLB without taking the size of p into account.

As a second example, consider:

`pc=here+2; #4; pc=here+2; #2; !;` p `; ##pc`

Fragment p is invoked from different locations, returning control to wherever it was called from. PGLBpc allows *one-shot* invocation (no recursion, no chained invocation). In PGLBpc it is possible to use pre-compiled libraries without altering them in order to link. Only their size is needed, when more than one is used, to establish their start as a relative offset from the main program.

3.1 Projection from PGLBpc to PGLB

The projection from PGLBpc to PGLB is defined as follows:

$$\mathit{pglbpc2pglb}(u_1;\ldots;u_n) = \psi_p(\emptyset, \phi_1^n(u_1);\ldots;\phi_n^n(u_n);\#0;\#0)$$

An instruction #0 ensures that a program reaching its end without terminating doesn't "fall through" in the jump table which is placed there in ψ_p. An additional #0 is added in case u_n is conditional. Jumps beyond the boundaries of the program are adapted for the same reason. Here, ϕ_i^n and ψ_p are defined as follows:

- $\phi_i^n(\#k) = \#0$ when $i+k > n$;

- $\phi_i^n($`pc=here+`$k) = u$, where $u \equiv$ `pc.set:`i`+`k when $i+k > n$, and $u \equiv \#0$, otherwise;

- $\phi_i^n(\backslash\#k) = \#0$ when $k \geq i$;
- $\phi_i^n(\texttt{pc=here}{-}k) = u$, where $u \equiv \texttt{pc.set:}i{-}k$ when $k \geq i$, and $u \equiv \#0$, otherwise;
- $\phi_i^n(\texttt{pc=here}) = \texttt{pc.set:}i$;
- $\phi_i^n(\#\#\texttt{pc}) = \#n - i + 3$;
- $\phi_i^n(u) = u$, otherwise;
- $\psi_p(S, u_1; \ldots; u_m) = \psi_p(S \cup \{l\}, u_1; \ldots; u_m; {+}\texttt{pc.eq:}l; \backslash\#m{-}l{+}2)$ when $l \notin S$ and for some j $(1 \leq j \leq m)$ we have $u_j = \texttt{pc.set:}l$;
- $\psi_p(S, p) = p$ otherwise.

The image of a PGLBpc program is assumed to use state machine `smpc` via focus `pc`.

Projection *pglbpc2pglb* is not the identity on PGLB (certain instructions $\#k$ and $\backslash\#k$ are mapped to #0). However, *pglbpc2pglb* can still be regarded as a conservative extension in that it attributes the same meaning to PGLB programs: $\mathit{pglb2pgla}(\mathit{pglbpc2pglb}(p)) = \mathit{pglb2pgla}(p)$ (*pglb2pgla* maps precisely the same instructions $\#k$ and $\backslash\#k$ to #0).

Looking at our earlier examples (assuming the size of p is n_p, and assuming p to be mapped to $\widehat{p}$):

pglbpc2pglb(`pc=here+1;` p `; +a; ##pc`) =
`pc.set:2;` $\widehat{p}$ `; +a; #3; #0; #0; +pc.eq:2; \#`n_p`+5`

pglbpc2pglb(`pc=here+2; #4; pc=here+2; #2; !;` p `; ##pc`) =
`pc.set:3;#4;pc.set:5;#2;!;`$\widehat{p}$`;#3;#0;#0;+pc.eq:3;\#`$7{+}n_p$`;+pc.eq:5;\#`$7{+}n_p$

4 Absolute addressing with explicit PC

PGLD is the PGA language with absolute addressing. It is relevant as a model for many processors, which have absolute addressing as their most basic control mechanism. It is also relevant in that it offers a very straightforward target for projections from other PGA languages.

PGLDpc is an extension of PGLD with explicit PC. In addition to PGLD's $\#\#k$ (absolute goto), PGLDpc offers the following control instructions:

- `pc=`k
  Set the value PC to k (the location of the k-th instruction), or terminate if no such instruction exists;
- `##pc`
  Proceed execution at the instruction referred to by the program counter, if that exists, or terminate, otherwise.

Like PGLD, the significance of PGLDpc is that it offers a straightforward target for projections. Other than that PGLDpc exhibits no significant properties: the capabilities it adds are similar to those added by PGLBpc, except that they are less useful (templates abstracting from fragment size mean little if the absolute location of every instruction is to be incorporated). PGLDpc does have one shot invocation.

By way of an example, consider the following two programs, which are the PGLDpc versions of the examples from the previous section.

`pc=2;` p `; +a; ##pc`

`pc=3; ##6; pc=5; ##6; !;` p `; ##pc`

4.1 The projection of PGLDpc

Two routes to define the meaning of PGLDpc are appropriate. PGLD is given a meaning in a projection to PGLC, which again derives its meaning from a projection to PGLB. Both projections are straightforward. Based upon these, PGLCpc could easily be defined, and conservative extensions of the respective maps from PGLDpc to PGLCpc and from there to PGLBpc.

Alternatively, a projection from PGLDpc to PGLD could be defined in the spirit of *pglbpc2pglb*.

We chose the first alternative because it strengthens the view that an explicit PC is an orthogonal extension to PGA languages. In addition, the jump table introduced by *pglbpc2pglb* is an artifact introduced to express the fact that in computer hardware the program counter can be manipulated as data, whereas in PGA the abstract value PC and the locus of execution live in different realms altogether. It seems appropriate to express this fact only once. Note that we suggest nor deny that PGLCpc serves any significant purpose on its own.

4.1.1 PGLCpc

PGLCpc adds the same advanced control instructions to PGLC as does PGLBpc to PGLB (`##pc`, `pc=here`, `pc=here+`k, `pc=here-`k). Its meaning is defined by *pglc2pglb*, the projection of PGLC to PGLB, under the understanding that the domain of that projection has been extended with the instructions above; the projection is the identity on that extension by the third clause of ϕ_j^k (see below). For completeness, we repeat the definition of *pglc2pglb* from [BL02].

$$pglc2pglb(u_1;\ldots;u_k) = \psi_1^k(u_1);\ldots;\psi_k^k(u_k);!;!$$

Here:

- $\psi_j^k(\#l) =$! when $j + l > k$;
- $\psi_j^k(\backslash\#l) =$! when $l \geq j$;
- $\psi_j^k(u) = u$, otherwise;

4.1.2 The projection from PGLDpc to PGLCpc

The projection from PGLDpc to PGLCpc is defined as follows:

$$\mathit{pgldpc2pglcpc}(u_1;\ldots;u_k) = \psi_1(u_1);\ldots;\psi_n(u_n)$$

Here:

- $\psi_j(\#\#l) = \#l - j$ when $l \geq j$;
- $\psi_j(\#\#l) = \backslash\#j - l$ when $l < j$;
- $\psi_j(\texttt{pc=}l) = \texttt{pc=here+}l - j$ when $l > j$;
- $\psi_j(\texttt{pc=}l) = \texttt{pc=here}$ when $l = j$;
- $\psi_j(\texttt{pc=}l) = \texttt{pc=here-}(l - j)$ when $l < j$;
- $\psi_j(u) = u$, otherwise;

With the exception of the clauses handling PC-related instructions, the definition of *pgldpc2pglcpc* is identical to that of *pgld2pglc*. Accordingly, *pgldpc2pglcpc* is a proper extension.

Obviously the image of our examples under *pgldpc2pglcpc*, disregarding the projection of fragment p itself, is precisely as they appeared in Section 3 .

5 Structured programming

Every imperative "higher" programming language favors so-called *structured programming*. At the basis is the *block*: an arbitrarily large program fragment that behaves as a unit in certain circumstances. A block can be entered, in which case it is processed normally, or it can be passed over completely. Blocks can be nested, and they are demarcated with matching braces, as is quite common.

Structured programming is obviously relevant to a human programmer; keeping track of absolute locations, and having to change half of them when the program is altered is error prone and cumbersome.

In addition structured programming is relevant in projections in that it brings back location independence and monotony which were lost in PGLD.

In PGLDS, the goto instruction ($\#\#k$) is removed because it is in conflict with the idea of location and size independence. We do not suggest that the instruction is superfluous; see [BL02] for a proof to the contrary, under similar circumstances.

PGLDS has the control instructions shown below.

- { and } mark the beginning and end of blocks. When executed they do nothing other than advancing to the next instruction (or terminate if no such instruction exists);

- #> passes control two instructions after the next block, or, whichever comes first, after the next } if the matching { occurs before #> or is missing altogether, or terminates if none of these apply. That is, control passes after the next block, the enclosing block, or the entire program, whichever comes first;
- <# passes control two instructions before the immediately preceding block, or, whichever comes first, before the last { if the matching } occurs after <# or is missing altogether, or terminates if none of these apply. That is, control passes before the previous block, the enclosing block, or the entire program, whichever is closest;
- ∗`skip` advances to the next instruction, terminating if no such instruction exists.
- **!** terminate execution.

The instructions immediately preceding { or following } are limited to termination, skip and jump instructions.

Some examples may help to illustrate this. Unless specified otherwise, fragments "..." may contain braces, but only in matching pairs.

```
-a; #>; {; ... }; *skip
```

If basic instruction `a` yields true, control passes to the opening brace, and the block is processed. Otherwise, control passes to #> and then immediately after the ∗`skip`. That is, this is a template for the statement "`if (a) { ... }`" (in Java syntax).

```
-a; #>; {; ... }; #>; {; ... }; *skip
```

This template implements "`if (a) { ... } else { ... }`".

```
-a; #>; {; ... }; <#
```

This template implements "`while (a) { ... }`".

```
-a; #>; {; ... #>; ... <#; ...}; <#
```

Again, a while loop. The enclosed #> and <# behave as `break` and `continue` statements, respectively, if the ellipses do not contain any block.

5.1 Projection from PGLDS to PGLD

The projection from PGLDS to PGLD giving a meaning to a program $p = u_1; \ldots; u_n$ is defined as follows.

$$\mathit{pglds2pgld}(u_1; \ldots; u_n) = \phi_{1,p}(u_1); \ldots; \phi_{n,p}(u_n)$$

Here, $\phi_{i,p}$, and $l^<$ and $l^>$ (used by ϕ) are defined as follows:

- $\phi_{i,p}(!) = \#\#0;$

- $\phi_{i,p}(u) = \#\#i+1$ for $u \equiv *\texttt{skip}$, $u \equiv \{$ or $u \equiv\}$;
- $\phi_{i,p}(\#>) = \#\#l^{>}(0,i,p)$;
- $\phi_{i,p}(<\#) = \#\#l^{<}(0,i,p)$;
- $\phi_{i,p}(u) = u$, otherwise;
- $l^{>}(k,i,u_1;\ldots;u_n) =$
  - 0 when $i > n$;
  - $l^{>}(k+1,i+1,u_1;\ldots;u_n)$ when $u_i \equiv \{$;
  - t when $u_i \equiv\}$ and $k \leq 1$, where $t = i+2$ if $i+2 \leq n$ and $t = 0$, otherwise;
  - $l^{>}(k-1,i+1,u_1;\ldots;u_n)$ when $u_i \equiv\}$ and $k > 1$;
  - $l^{>}(k,i+1,u_1;\ldots;u_n)$, otherwise;
- $l^{<}(k,i,u_1;\ldots;u_n) =$
  - 0 when $i \leq 0$;
  - $l^{<}(k+1,i-1,u_1;\ldots;u_n)$ when $u_i \equiv\}$;
  - t when $u_i \equiv \{$ and $k \leq 1$, where $t = i-2$ if $i-2 \geq 1$ and $t = 0$, otherwise;
  - $l^{<}(k-1,i-1,u_1;\ldots;u_n)$ when $u_i \equiv \{$ and $k > 1$;
  - $l^{<}(k,i-1,u_1;\ldots;u_n)$, otherwise;

The images under *pglds2pgld* of our examples are as shown below. Here, the lengths of the fragments indicated by ellipses are taken to be 1 for the computation of absolute addresses.

pglds2pgld(`-a; #>; {; ... }; *skip`) = `-a; ##7; ##4; ... ##6; ##7; ...`

pglds2pgld(`-a; #>; {; ... }; #>; {; ... }; *skip`) =
`-a; ##7; ##4; ... ##6; ##11; ##8; ... ##10; ##11; ...`

pglds2pgld(`-a; #>; {; ... }; <#`) = `-a; ##7; ##4; ... ##6; ##1; ...`

pglds2pgld(`-a; #>; {; ... #>; ... <#; ...}; <#`) =
`-a; ##11; ##4; ... ##11; ... ##1; ... ##10; ##1; ...`

6 Structured programming with exceptions

One of the accepted deficiencies of purely structured programming is exception handling. In most programming languages exception handling is absent or rudimentary; in some languages, such as Lisp, ADA and Java, it is very refined and expressive.

Key to most if not all mechanisms is that an exception handler is made known ("catch", "try", ...) before any exception can occur, and that it will handle any subsequent exception, possibly of a specific type, which occurs either implicitly in the run time system or explicitly by user control ("throw", "raise", ...).

PGLDSpc, the extension of PGLDS with an explicit PC, offers precisely such a mechanism.

PGLDSpc adds the following instructions to PGLDS:

- `pc=here>`
  set the value PC to the location #> would jump to, from the current location (or terminate if that is outside the program);
- `pc=<here`
  set the value PC to the location <# would jump to, from the current location;
- ##`pc`
  Proceed execution at the instruction referred to by the program counter, if that exists, or terminate, otherwise.

As an example, consider the following ("..." may contain braces, but only in matching pairs).

```
#>; *skip; *skip; {; h; }; *skip; pc=<here; ... ##pc; ...
```

The fragment h is skipped entirely, but PC is set to the location of its beginning. The fragment is invoked by executing ##`pc`.

6.1 Projection from PGLDSpc to PGLDpc

The projection from PGLDSpc to PGLDpc of a PGLDSpc program $p = u_1; \ldots; u_n$ is a conservative extension of *pglds2pgld* defined as follows.

$$\mathit{pgldspc2pgldpc}(u_1; \ldots; u_n) = \phi_{1,p}(u_1); \ldots; \phi_{n,p}(u_n)$$

Here, $\phi_{i,p}$ is defined as follows ($l^<$ and $l^>$ are defined as before).

- $\phi_{i,p}(!) = \#\#0$;
- $\phi_{i,p}(u) = \#\#i + 1$ for $u \equiv$ `*skip`, $u \equiv$ { or $u \equiv$};
- $\phi_{i,p}(\#>) = \#\#l^>(0, i, p)$;

- $\phi_{i,p}(\texttt{pc=here>}) = \texttt{pc=}l^{>}(0,i,p)$;
- $\phi_{i,p}(\texttt{<\#}) = \texttt{\#\#}l^{<}(0,i,p)$;
- $\phi_{i,p}(\texttt{pc=<here}) = \texttt{pc=}l^{<}(0,i,p)$;
- $\phi_{i,p}(u) = u$, otherwise;

The result of our example, assuming the size of h to be n_h and assuming h to be mapped to $\widehat{h}$, is shown below:

pglv2pgle(`#>; *skip; *skip;` {`;` h`; };` `*skip; pc=<here; ... ##pc; ...`) =
`##`7+n_h`; ##3; ##4; ##5;` $\widehat{h}$`; ##`6+n_h`; ##`7+n_h`; pc=2; ...; ##pc; ...`

7 Vector Labels

Structured programming concerns the flow of control under normal circumstances, pursuant to the structure of a program. This is too limited when considering exceptional circumstances or loosely related program fragments such as modules. As mentioned, PGLDSpc improves upon this situation in a very limited way by introducing a single type of exception.

In a sense, the other extreme of a range is PGLE: it allows for arbitrarily complex control flow including multiple exceptions and other mechanisms, unrelated to the program structure.

Under certain conditions PGLE is monotonous. These conditions avoid unintended reuse of the same label in distinct program fragments that are to be combined. In practice, this means that fragments which are to be combined have to be checked to compare labels, and have to be modified by replacing labels when the same label is defined in both, taking care that the labels that are introduced do not clash with any label already occurring in the fragments or in their context. Often, projections become significantly more complex in order to handle this properly.

PGLV offers a partial solution to this problem by using vectors of numbers rather than singleton numbers. Then, program fragments can be combined, for example, by prepending all labels defined and used in the n-th fragment with the number n (a label being used but not defined might, depending upon the situation, be an error or a reference to another fragment; how to handle this depends on that situation). Since vectors have arbitrary size, this process can be repeated indefinitely without a clash ever being introduced inadvertently.

For all clarity, PGLV is monotonous under similar conditions as PGLE. However, PGLV offers a mechanism which makes guaranteeing these conditions significantly more easy.

A vector label will appear as $[n_1, \ldots, n_k]$.

In PGLV, the following control instructions replace the corresponding control instructions in PGLE.

- $[n_1, \ldots, n_k]$, where $n_k \neq 0$ (replaces $£k$)
  Define the given label. When executed, do nothing other than advancing to the next instruction, or terminate, if no such instruction exists. Trailing zeroes are excluded for the reason given in Section 7.1;

- $\#\#[n_1, \ldots, n_k]$ (replaces $\#\#£k$)
  Continue execution at the first location in the program where the given label is defined, or terminate, when no such location exists;

It is not illegal to refer to (as opposed to define) labels with trailing zeroes (it's just confusing).

Like PGLE, PGLV requires the instruction immediately following a conditional basic instruction to be either **!** or a goto instruction.

The following is a PGLV program:

```
+a; ##[1,2]; ##[3,4,5]; [1,2]; b; !; [3,4,5]; c
```

7.1 Vectors and map $\langle \ldots \rangle$

The function $\pi(n, m) = \frac{1}{2}(m^2 + 2mn + n^2 + 3m + n)$ is an bijective map from $\mathbb{N} \mapsto \mathbb{N} \times \mathbb{N}$ (the points in the natural grid are visited in successive diagonals). Using π, a pair of numbers can be encoded in one number.

This function can be used to encode a vector of numbers in one number. Let $\langle n \rangle = \pi(n, 0)$ and $\langle n_1, \ldots, n_k \rangle = \pi(n_1, \langle n_2, \ldots, n_k \rangle)$. Note that every number encodes infinitely many vectors. For example, $0 = \langle 0 \rangle = \langle 0, 0 \rangle = \ldots$ and so forth. Indeed, $\langle n_1, \ldots, n_k \rangle = \langle n_1, \ldots, n_k, 0 \rangle$. When trailing zeroes are excluded, the encoding is unique.

The vector encoded by a number p can be found as follows. Let $d = \frac{1}{2}(\sqrt{8p+1} - 1)$ (d for diagonal, since p occurs on the d-th diagonal), let i be the integral part of d, and let $m = p - \frac{1}{2}i^2 - \frac{1}{2}i$. Now, $p = \langle d - m, m \rangle$. The entire vector encoded by p is found by repeating this step until $m = 0$.

7.2 Projection from PGLV to PGLE

The projection of PGLV is defined as follows:

$$\mathit{pglv2pgle}(u_1; \ldots; u_n) = \phi(u_1); \ldots; \phi(u_n)$$

Function ϕ is defined below, where map $\langle \ldots \rangle$ will be defined subsequently.

- $\phi([n_1, \ldots, n_k]) = £v$ where $v = \langle n_1, \ldots, n_k \rangle$;
- $\phi(\#\#[n_1, \ldots, n_k]) = \#\#£v$ where $v = \langle n_1, \ldots, n_k \rangle$;
- $\phi(u) = u$, otherwise.

Applied to our earlier example, this means:

pglv2pgle(`+a; ##[1,2]; ##[3,4,5]; [1,2]; b; !; [3,4,5]; c`) =
`+a; ##L8; ##L1481; L8; b; !; L1481; c`

8 A higher programming language

Most higher programming languages combine structured programming with the ability to express non-structural behavior. Within a module, and under normal circumstances, structured programming is favored, but between modules or when exceptions occur, mechanisms which are not structure-oriented are favored. PGLVS, the union of PGLV and PGLDS, exhibits precisely this combination.

As in PGLE, the instruction immediately following a conditional is limited to termination (!) and jump instructions (##[$n_1, \ldots, n_k$], #> and <#). For similar reasons, the instructions immediately before { and after } are limited to skip, jump and termination instructions.

Example:

```
-a; #>; {; ..
   -b; #> {; .. +c; ##[1]; .. +d; ##[2]; .. e; ..
        #>  [2]; h2;
   }; <#; ..
}; !; !; [1]; h1; !
```

This example contains a nested loop. The outer loop is performed while `a` is true; the inner loop while `b` is true. The fragment containing `e` is performed in the inner loop, unless exception `c` occurs, in which case both loops are left and handler `h1` is applied, or exception `d` occurs, in which case the inner loop is left, but the outer loop is continued after handler `h2` finishes.

Under mild conditions PGLVS is again monotonous. The most notable conditions are concerned with name clashes (the same label being used independently in multiple fragments) and well-formedness regarding { and }.

8.1 The projection of PGLVS

PGLE is a restriction on the language PGLDg and derives its meaning from the projection *pgldg2pgld* defined in [BL02], shown here for clarity (we deviate slightly from the original notation):

$$\mathit{pgldg2pgld}(u_1;\ldots;u_n) = \psi_1(u_1);\ldots;\psi_{n,p}(u_n)$$

Here, $\psi_{i,p}$, and auxiliary functions are defined as follows:

- $\psi_{i,p}(!) = \#\#0$;
- $\psi_{i,p}(\#\#\pounds k) = \#\#\tau_p(k)$;
- $\psi_{i,p}(\pounds k) = \#\#i{+}1$;
- $\psi_{i,p}(u) = u$, otherwise;
- $\tau_p(k) = j$, when $p = u_1;\ldots;u_n$ and j is the least number such that $u_j \equiv \pounds k$, or $j = 0$ when $\pounds k$ is not defined in p.

Now, let *pglv2pgld* be defined by $pglv2pgld(p) = pgldg2pgld(pglv2pgle(p))$. The projection *pglvs2pgld* from PGLVS is defined by the composition of *pglv2pgld* and *pglds2pgld* under the following understanding:

- The domain of *pgldg2pgld* is extended to contain PGLDS's structured primitives and PGLD's absolute goto. On this extension, *pgldg2pgld* is the identity (by the fourth clause of $\psi_{i,p}$);
- The domain of *pglds2pgld* is extended to contain PGLV's label-related instructions and PGLD's absolute goto. On this extension, *pglds2pgld* is the identity (by the fifth clause of $\phi_{i,p}$);

Under these assumptions, that composition is in fact commutative, as is easily verified. Note that to that end PGLD's absolute goto is added to both respective domains (it is not contained in PGLVS, however).

Applying this projection to our earlier example results in the following. Here the size of fragments `..` is taken to be 1 for the computation of all absolute addresses.

pglvs2pgld`(-a;#>; {; .. -b; #>; {; .. +c; ##[1]; .. +d; ##[2]; .. e; ..`
`#>[2]; h2; }; <#; .. }; !; [1]; h1; !)` =
```
-a; ##25; ##4; .. -b; ##22; ##8; .. +c; ##26; .. +d; ##18; .. e; ..
##22; ##19; h2; ##21; ##5; .. ##24; ##0; ##0; ##27; h1; ##0
```

9 Invocation and returning goto

The invocation of a procedure, function, method, etcetera (we shall use the word *method* to signify all these forms), directs processing from *its* location to that of the instructions 'belonging to' the method. When the invoked method has fulfilled its purpose, processing resumes immediately after the invocation. Since the same method can be invoked from different locations, that location must somehow be remembered until processing resumes. Since an invoked method may invoke other methods, the number of loci that must be remembered is, a priori, unbounded.

The mechanism that implements invocation is the *returning goto* (or *call*): like a plain *goto* it directs processing of instructions to continue elsewhere, but in addition, the location following the returning goto instruction is remembered. The *return* instruction directs processing to continue immediately after the last returning goto.

In [BB02] this mechanism is defined using primitives which are not present in our current framework (notably: molecular programming primitives). In consequence, we redefine this mechanism using primitives presented here.

PGLI is the PGA-language with method invocation. In addition to the control instructions of PGLVS, it offers the following control instructions:

- $R\#\#[n_1, \ldots, n_k]$
  Go to the first location in the program where the given label is defined, and save the current location for future use; if no such label exists, terminate;

- $\#\#R$
  Return to the last saved location; if none was saved, terminate.

For example,

```
a; R##[1]; b; !; [1]; c; ##R
```

An extended example appears in Section 11.

We will now proceed as follows.

Remembering multiple previous locations of execution requires the aid of a new state machine. First we will define a state machine for returning goto and we will show that it is projection compatible. Then we will define PGLVSpc, the extension of PGLVS with explicit PC. Finally, we will present the projection of PGLI.

9.1 The state machine for returning goto

Earlier we mentioned that a state machine is projection compatible on a class of programs if every program in that class can only make it reach finitely many states. Unrestricted use of returning goto can lead to non-trivial infinite recursion (which is not finitely representable) even for simple programs.

In consequence we limit the state machine. The call-once state machine `smco` remembers an address only once; it allows for multiple methods being called, and methods being called from multiple locations, but it does not allow for (mutual) recursion. A priori this state machine is again unbounded, but now only finitely many states can be reached for any given program, as we will make clear.

The state machine `smco` $= \langle \Sigma_{co}, F_{co} \rangle$ where $\Sigma_{co} = \cup_{i \in N} \{\texttt{set:}i, \texttt{eq:}i\} \cup \{\texttt{push}, \texttt{pop}\}$ and $F_{co}(\sigma) = \mathit{true}$ precisely when $\sigma \equiv \rho; \texttt{eq:}k$, when ρ contains at least one `set` instruction, when the first `push` instruction (if any) is preceded by a set instruction, when $\psi(\rho, 0, mt()) = \langle k, S \rangle$ and when $S \neq err()$, and it is *false* in any other situation.
Here, ψ is defined as follows:

- $\psi(\texttt{set:}k; \sigma, n, S) = \psi(\sigma, k, S)$;
- $\psi(\texttt{push}; \sigma, n, S) = \psi(\sigma, n, push(n, S))$;
- $\psi(\texttt{pop}; \sigma, n, push(k, S)) = \psi(\sigma, k, S)$;
- $\psi(\texttt{pop}; \sigma, n, mt()) = \psi(\sigma, 0, err())$;
- $\psi(v; \sigma, n, S) = \psi(\sigma, n, S)$ for any other instruction v;
- $\psi(\sigma, n, S) = \langle n, S \rangle$ when σ is the empty sequence.

The third argument of ψ is an element of the algebra S_{ue} of *unique element stacks*, which has signature $\{mt() \mapsto S_{ue}, push(\mathbb{N}, S_{ue}) \mapsto S_{ue}, err() \mapsto S_{ue}\}$ (*mt*, when pronounced, sounds like *empty*). In this algebra, the following equalities hold: $push(n, err()) = err()$ and $push(n, S) = err()$ when S already holds n.

9.2 Projection compatibility of `smco`

We discuss projection compatibility of `smco` on PGLD because it implies projection compatibility for all PGA languages, and PGLD is a straightforward target, for our projection.

Given a program $p = u_1; \ldots; u_m$, let L be the set of values occurring in a `set` or `eq` instruction for focus `pc`, let $\widehat{S}$ be the set of non-empty sequences (tuples) of elements of L in which each element of L occurs at most once, and let $S = \{s_1, \ldots, s_k\} = (\widehat{L} \times \widehat{S}) \cup \{\langle x_1 \rangle, \langle y_1 \rangle\}$, where x_1 and y_1 are unequal numbers not occurring in L ($\langle x_1 \rangle$ will represent the machine's state before any value has been set, and $\langle y_1 \rangle$ will represent the machines state after a number has been pushed more than once or after underflow occurs). We use the notation $\langle k, l, \vec{m} \rangle$ to signify a tuple with elements k, l, and zero or more remaining elements $\vec{m}$, and the notation $|s|$ signifying the number of elements in tuple s. Tuples are taken to be elements of the Cartesian product, so $\alpha \times \langle \vec{\beta} \rangle \equiv \langle \alpha, \vec{\beta} \rangle$ We assume $s_1 = \langle x_1 \rangle$ and $s_2 = \langle y_1 \rangle$.

The projection of a *pgld*::*pc-co* program $p \equiv u_1; \ldots; u_m$ with focus *pc* and using methods from Σ_{co} to a program not using that or a substitute state machine with equivalent behavior is defined as follows.

$$pgld\text{::}pc\text{-}co2pgld(p) = \phi_k^{m,S}(p)$$

Here, $\phi_i^{m,S}$ and auxiliary functions are defined as follows. Note that $l^m(i,j)$ signifies the location of the j-th instruction in fragment i.

- $\phi_1^{m,S}(u_1; \ldots; u_m) = \psi_{1,1}^{m,S}(u_1); \ldots; \psi_{1,m}^{m,S}(u_m);$
- $\phi_{i+1}^{m,S}(u_1; \ldots; u_m) = \phi_i^{m,S}(u_1; \ldots; u_m); \#\#0; \#\#0; \psi_{i+1,1}^{m,S}(u_1); \ldots; \psi_{i+1,m}^{m,S}(u_m);$
- $\psi_{i,j}^{m,S}(\#\#l) = \#\#t$ where $t = l^m(i,l)$ for $1 \leq l \leq m$, and $t = 0$, otherwise;
- $\psi_{i,j}^{m,S}(+\texttt{pc.set:}x) = \#\#t$ when $s_i \equiv \langle y, \vec{z} \rangle$, where $s_n \equiv \langle x, \vec{z} \rangle$ and $t = l^m(n,j)+2$ when $i \neq 2$, and $t = l^m(2,j)+2$, otherwise;
- $\psi_{i,j}^{m,S}(u) = \#\#t$ when $u \equiv \texttt{pc.set:}x$ or $u \equiv -\texttt{pc.set:}x$, when $s_i \equiv \langle y, \vec{z} \rangle$,where $s_n \equiv \langle x, \vec{z} \rangle$ and $t = l^m(n,j)+1$ when $i \neq 2$, and $t = l^m(2,j)+1$, otherwise;
- $\psi_{i,j}^{m,S}(+\texttt{pc.push}) = \#\#t$ when $s_i \equiv \langle x, \vec{z} \rangle$, where $s_n \equiv \langle x, x, \vec{z} \rangle$ and $t = l^m(n,j)+2$ when $i > 2$ and $x \notin \vec{z}$, and $t = l^m(2,j)+2$, otherwise;

- $\psi_{i,j}^{m,S}(u) = \#\#t$ when $u \equiv$ `pc.push` or $u \equiv -$`pc.push`, when $s_i \equiv \langle x, \vec{z} \rangle$, where $s_n \equiv \langle x, x, \vec{z} \rangle$ and $t = l^m(n,j)+1$ when $i > 2$ and $x \notin \vec{z}$, and $t = l^m(2,j)+1$, otherwise;

- $\psi_{i,j}^{m,S}(+$`pc.pop`$) = \#\#t$ where $s_i \equiv \langle x, \vec{z} \rangle$, where $s_n \equiv \langle \vec{z} \rangle$ and where $t = l^m(n,j)+2$ when $i > 2$ and $|s_i| \geq 2$, and where $t = l^m(2,j)+2$, otherwise;

- $\psi_{i,j}^{m,S}(u) = \#\#t$ when $u \equiv$ `pc.pop` or $u \equiv -$`pc.pop`, where $s_i \equiv \langle x, \vec{z} \rangle$, where $s_n \equiv \langle \vec{z} \rangle$ and where $t = l^m(n,j)+1$ when $i > 2$ and $|s_i| \geq 2$, and where $t = l^m(2,j)+1$, otherwise;

- $\psi_{i,j}^{m,S}(u) = \#\#l^m(i,j)+1$ when $i > 2$ and when $u \equiv +$`pc.eq:`x and $s_i \equiv \langle x, \vec{z} \rangle$, or when $u \equiv -$`pc.eq:`x and $s_i \equiv \langle y, \vec{z} \rangle$, and $x \neq y$, or when $u \equiv$ `pc.eq:`x;

- $\psi_{i,j}^{m,S}(u) = \#\#l^m(i,j)+2$ when $i \leq 2$ or when $u \equiv -$`pc.eq:`x and $s_i \equiv \langle x, \vec{z} \rangle$, or when $u \equiv +$`pc.eq:`x and $s_i \equiv \langle y, \vec{z} \rangle$, and $x \neq y$;

- $\psi_{i,j}^{m,S}(u) = u$, otherwise;

- $l^m(i,j) = (i-1) * (m+2) + j$.

We conjecture that the behavior of a program in combination with `smco` is identical to that of the projected program; i.e., that given a PGLD::pc-co program p, $|p|_{\text{PGLD::pc-co}} /_{pc}$ `smco` $\equiv_{be} |$*pgld::pc-co2pgld*$(p)|_{\text{PGLD}}$.

As an example of this projection, consider the following. For clarity we have prepended each line with the tuple related to the fragment in that line, and we have subscripted some instructions with their location.

pgld::pc-co2pgld(`pc.set:1; pc.push; pc.pop; pc.pop; !`) =
$\langle x_1 \rangle$ `##16`$_1$`; ##10`$_2$`; ##11`$_3$`; ##12`$_4$`; !`$_5$`; ##0; ##0;`
$\langle x_2 \rangle$ `##9`$_8$`; ##10`$_9$`; ##11`$_{10}$`; ##12`$_{11}$`; !`$_{12}$`; ##0; ##0;`
$\langle 1 \rangle$ `##16`$_{15}$`; ##24`$_{16}$`; ##11`$_{17}$`; ##12`$_{18}$`; !`$_{19}$`; ##0; ##0;`
$\langle 1,1 \rangle$ `##23`$_{22}$`; ##10`$_{23}$`; ##18`$_{24}$`; ##19`$_{25}$`; !`$_{26}$

9.3 PGLVSpc

PGLVSpc is the extension of PGLVS with explicit PC. It is the union of PGLVS and PGLDSpc with the single additional instruction `pc` $= [n_1, \ldots n_k]$.

The projection *pglvspc2pgldpc* of PGLVSpc is defined as follows. First, given a PGLVSpc program $p \equiv u_1; \ldots; u_m$, let

$$pcv(p) = \phi_p(u_1; \ldots; u_m)$$

Here, ϕ_p is defined as follows:

- $\phi_p($`pc=`$[n_1, \ldots, n_k]) =$ `pc=`j where j is the first location with the instruction $[n_1, \ldots, n_k]$, if that exists, and $j = 0$, otherwise;

- $\phi_p(u) = u$, otherwise.

Now, *pglvspc2pgldpc* is the composition of *pglvs2pgld*, *pgldspc2pgldpc* and *pcv*, under the assumption that all domains have been extended appropriately and that all projections are the identity on these extensions. Under this assumption the composition commutes.

That this definition makes sense can be seen as follows. Intuitively, one verifies easily that if the PGLD-meaning of the instruction $\#\#[n_1, \ldots, n_k] = j$, in some context, then the PGLDpc-meaning of the instruction $pc{=}[n_1, \ldots, n_k] = j$, as one would expect.

More thoroughly one should observe that all relevant projections map any instruction precisely to one instruction, which implies that the location of a label definition in a PGLVSpc program is the same as its location in the PGLD-meaning of that program.

As an example consider the following. For absolute address calculations the size of the fragment `...` is taken to be one.

pglvspc2pgldpc(`pc=here>; {; ... }; pc=[1,2,3]; [1,2,3]; *skip`)) =
`pc=6; ##3; ...; ##5; pc=6; ##7; ##8`

9.4 Projection from PGLI

Given a PGLI program $p = u_1; \ldots; u_n$, its projection to PGLVSpc is defined as follows:

$$pgli2pglvspc(u_1; \ldots; u_n) = \phi(u_1); \ldots; \phi(u_n)$$

Here, ϕ is defined as follows:

- $\phi(R\#\#[n_1, \ldots, n_k]) =$ {; `pc=here>; pc.push;` }; $\#\#[n_1, \ldots, n_k]$;
- $\phi(\#\#R) =$ `pc.pop;` $\#\#$`pc`;
- $\phi_{i,p}(u) = u$, otherwise;

This projection assumes that the program uses the state machine `smco` via focus `pc`.

Considering the earlier example, we have:

pgli2pglvspc(`a; R##[1]; b; !; [1]; c; ##R`)) =
`a; {; pc=here>; pc.push; }; ##[1]; b; !; [1]; c; pc.pop; ##pc`

10 Component Format and Flat File System

A file system contains a collection of files. Associated with each file are its contents, and additional information such as the name of the file. The name of a file identifies that file in a certain scope. If the collection of files is an otherwise

unstructured set, and if the scope within which a name identifies a file is the entire set, we speak of a *flat* file system[2].

It is common to describe the language or format of the contents of a file as a suffix of its name, sometimes called the extension. We shall refer to this as a file's *form*.

Often, file names are arbitrary size-bounded strings over some alphabet, but we abstract from this as follows. File names are of the form `c`i`:FORM`, for component i ($i \in I\!N$) with form `FORM`. For example, `c0:pgle`.

In practice, further information is kept with a file, pertaining to ownership and access rights, physical location on the underlying media, dates of modification, backup, etcetera. We will not go into this.

Also note that we are only interested in the behavior of programs consisting of multiple files. We do not consider files which represent anything other than programs, nor do we consider alterations to the file system. Arguably, we are being overly restrictive. For example, a compiler must, by definition, transgress both restrictions, and yet it falls well within our conceptual framework, in that it embodies a projection. Nonetheless, PGA focuses on control behavior rather than data manipulation, so files representing data, and the behavior of multi-file programs that alter their file system, remain significant areas which will get appropriate attention elsewhere.

To conclude, a file system F is a set of tuples $\prec n, p \succ$, where n is a name as described above, and p is a program with the appropriate form. For every name n at most one tuple occurs in F with that name. We will refer both to the tuple and to the contents of the associated file as "component", assuming that the reader can infer our intention. For convenience we define the functions $\mathit{lookup}(nm, F) = \prec nm, p \succ \in F$ if that component exists, and $\mathit{index}(n, f, F) = \mathit{lookup}(nm, F)$ for $nm = \mathtt{c}n\mathtt{:}f$.

10.1 PGLIcf

PGLIcf is the PGA-language for the *component format*: two advanced control instructions related to multiple components are added to PGLI: $\#\#\mathtt{c}i[n_1, \ldots, n_k]$, called the *non-local goto*, and $\mathtt{R}\#\#\mathtt{c}i[n_1, \ldots, n_k]$, called the *non-local invocation*.

The projection from PGLIcf not only replaces these control instructions, but ensures that all referenced components are included in its result[3].

Given a file system F and a component c in that file system, the projection from PGLIcf is defined as follows. Let $C = \{c_{i_1}, \ldots, c_{i_k}\}$ be the smallest set

[2]Without going into this further, we mention that in a *hierarchical* file system the collection is structured hierarchically as a set of files and other sets (called directories); the scope of a file name is the set in which the file is an element; and a file is therefore identified by its name and a *path*: an identification of one directory in the hierarchy of sets.

[3]A straightforward approach would be to concatenate all programs in the file system, making sure that the one program under consideration appears first, and replacing all non-local goto's with an appropriate local goto. Although formally sufficient, this approach deviates substantially from practice, where a *linker* recursively analyzes a program module to establish which other modules are required, and consequently produces the minimally sufficient program (obviously we do not regard dynamically linked modules here).

such that if some c_i ($i \in \{i_1, \ldots, i_k\}$) contains a reference to some c_j (i.e., an instruction $\#\#\texttt{c}j[m_1, \ldots, m_k]$ or $\texttt{R}\#\#\texttt{c}j[m_1, \ldots, m_k]$), and c_j is contained in F, then c_j is contained in C. We assume $c_1 = c$. Then,

$$\mathit{pglicf2pgli}(c, F) = \phi_C(C)$$

Here,

- $\phi_C(\prec nm_1, p_1 \succ, \ldots, \prec nm_k, p_k \succ) = \psi_1^C(p_1); \ldots; \psi_k^C(p_k)$;
- $\psi_n^C(u_1; \ldots; u_m) = \theta_n^C(u_1); \ldots; \theta_n^C(u_m); !; !$;
- $\theta_n^C(u) = !$, when u is a non-local goto or invocation (i.e., an instruction $\#\#\texttt{c}j[m_1, \ldots, m_k]$ or $\texttt{R}\#\#\texttt{c}j[m_1, \ldots, m_k]$) referring to a module not contained in C. Otherwise,
- $\theta_n^C(\#\#\texttt{c}i[n_1, \ldots, n_m]) = \#\#[i, n_1, \ldots, n_m]$;
- $\theta_n^C(\#\#[n_1, \ldots, n_m]) = \#\#[n, n_1, \ldots, n_m]$;
- $\theta_n^C(\texttt{R}\#\#\texttt{c}i[n_1, \ldots, n_m]) = \texttt{R}\#\#[i, n_1, \ldots, n_m]$;
- $\theta_n^C(\texttt{R}\#\#[n_1, \ldots, n_m]) = \texttt{R}\#\#[n, n_1, \ldots, n_m]$;
- $\theta_n^C([n_1, \ldots, n_m]) = [n, n_1, \ldots, n_m]$;
- $\theta_n^C(u) = u$, otherwise;

For example,

pglicf2pgli({$\prec$*c*1, `a; ##c2[1]`$\succ$, $\prec$*c*3, `[2]; b`$\succ$, $\prec$*c*2, `[1]; c`$\succ$}) =
`a; ##[2,1]; !; !; [2,1]; c; !; !`

11 Java

In this section we present an extended example which illustrates the following aspects:

- the projections developed in the previous section;
- the way in which it can be used to describe the behavior of a subset of Java sufficiently large to express the multi-file paradigm;
- our claim that PGA helps to reason about programs in that it explains unexpected program behavior.

Consider the following Java program, consisting of three classes. The "main" class file is `file:c0.java`(

```
class c0 {
   static void main(String s[]) {
      c1.m7();
   }
}
```

). It invokes a method in class file `file:c1.java`(

```
class c1 {
   static boolean b3 = c2.b5;
   static boolean b6 = true;
   static void m7() {
      if (b3) {
         b3=false;
         c2.b5=false;
         m7();
      } else {
         c2.m8();
      }
   }
}
```

), which again invokes a method in class file `file:c2.java`(

```
class c2 {
   static boolean b4 = c1.b6;
   static boolean b5 = true;
   static void m8() {
      System.out.println(b4);
      System.out.println(b5);
      System.out.println(c1.b6);
   }
}
```

).

The output of this program is twice "false", then "true", which may be somewhat surprising because seemingly variable b4 is set once, to a value which is demonstrated to be "true".

To express this program in the PGA family of languages we use instances of the state machine for boolean variables, as described in Section 1.2.

The projection from this subset of Java to PGLIcf is straightforward, with the following comments:

- PGA offers no data manipulation such as the assignment of one boolean variable to another boolean variable; this is implemented with a conditional construct. Note that we use the fact that a variable's initial value as defined in Section 1.2 is false;

- PGA programs have no I/O other than state changes in their co-program. In particular, there is no `println`, unless one would define a co-program offering that basic instruction. Since I/O is not the topic under investigation, we abstract from it, accepting the state of `smbv:4`, `smbv:5` and `smbv:6` after the program terminates as output;
- In Java initialization of static variables is implicit; in our program we have made this explicit by defining a "method" at label [1], in each module (except the 'main' module) that performs the initialization.

The following PGLIcf file system results: $f = \{$

$\prec 0,$

```
R##c2[1];
R##c1[1];
R##c1[7];
!
```

$\succ,$

$\prec 1,$

```
[1]
-smbv:5;
##[2];
smbv:3.set:true;
[2];
smbv:6.set:true;
##R;
[7];
-smbv:3;
##[3];
smbv:3.set:false;
smbv:5.set:false;
R##[7];
##[9];
[3];
R##c2[8];
[9];
##R
```

$\succ,$

$\prec 2,$

```
[1]
-smbv:6;
##[2];
smbv:4.set:true;
[2];
smbv:5.set:true;
##R;
[8];
*skip;
##R
```

$\succ\}$

Now, *pglicf2pgli*$(c0, f) =$

```
R##[2,1];
R##[1,1];
R##[1,7];
!;
!;
!;
[2,1];
-smbv:6;
##[2,2];
smbv:4.set:true;
[2,2];
smbv:5.set:true;
##R;
```

```
[2,8];
*skip;
##R;
!;
!;
[1,1];
-smbv:5;
##[1,2];
smbv:3.set:true;
[1,2];
smbv:6.set:true;
##R;
[1,7];
```

```
-smbv:3;
##[1,3];
smbv:3.set:false;
smbv:5.set:false;
R##[1,7];
##[1,9];
[1,3];
R##[2,8];
[1,9];
##R;
!;
!
```

We choose not to show further projections, because the amount of detail makes the example inaccessible. Instead we will discuss the program as listed above.

It is clear why `smbv:4` is false when the program terminates. It is set, if at all, only once, shortly after label [2,1], but only if `smbv:6` is true. However, label [2,1] is actually the second instruction processed, which implies that `smbv:6` is false at that time, by definition. Accordingly, `smbv:4` isn't set.

Naively, one could put forth that our projection from Java to PGLIcf is faulty: had component 1 been initialized before component 2, `smbv:4` would have gotten the right value. However, in that case `smbv:3` would have been set to the wrong value, leading to similarly unexpected output "true", "false", "true".

The order we have chosen is a consequence of the following mechanism, which is the one used in Java (with respect to this simple setting). While a module (i.e., class file) is being loaded, modules it refers to (i.e., uses) are identified. Before the module is initialized, any modules it refers to which are not yet loaded, are loaded and initialized. When this is done, initialization of the first module commences.

From this description we can see that the order in which modules are loaded, in our example, is `c0`, `c1`, `c2`, and that `c2` is initialized before `c1` is initialized. The reference from `c2` to `c1` doesn't matter, in this respect, since `c1` is already loaded when `c2` is initialized.

Where then does the initial value false come from, if not from initialization? When a module is loaded, but before it is initialized, every variable gets a default value which depends on its type. For booleans this default is false. Using state machines, the default value of a boolean variable is also false, so our implementation is a correct projection of the Java program.

12 Summary and conclusions

We have sketched a projection from a subset of Java which exhibits multi-file behavior in the sense described, to PGLIcf, a PGA-language which exhibits the

same feature. We have also defined a projection from that language to PGA defining the meaning of PGLIcf in every detail in terms of the program algebra for sequential programming. Based upon these projections we have given a "formal" meaning to the subset of Java described.

We have also shown how our approach can help to discuss complex program behavior by making explicit a feature which is handled implicitly in Java, thereby becoming inscrutable to the superficial user, possibly leading to unexpected results.

Finally, we have defined a number of useful languages in the context of PGA.

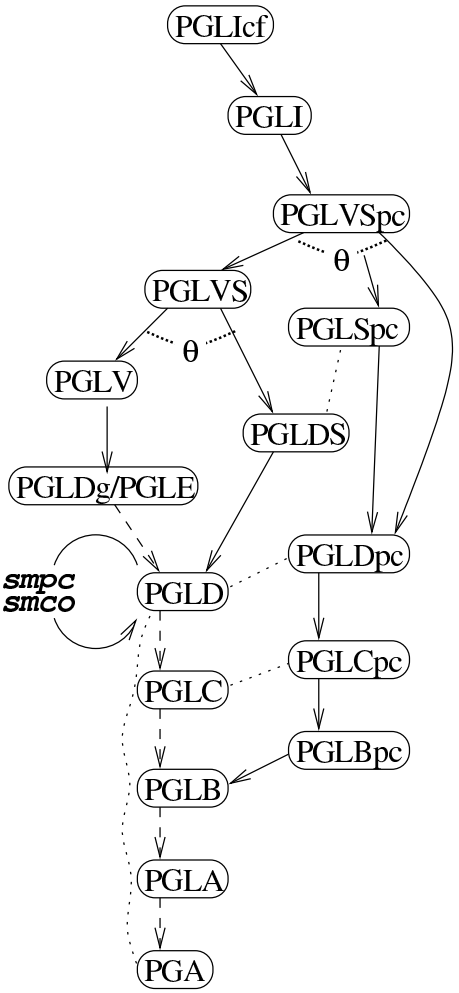
In the picture adjacent various languages and relevant projections have been depicted.

Solid arrows represent projections defined in this article; dashed arrows represent projections introduced elsewhere. Dotted lines signify embeddings where the related projections are conservative extensions.

The symbol θ signifies the definition of a projection as the commutative composition of projections shown.

The projections establishing compatibility of `smpc` and `smco` are shown, as is the embedding of PGA, PGLA, PGLB and PGLC in PGLD necessary to establish projection compatibility of those state machines in the entire framework.

As an overview we list key applications and predominant control instructions of most languages (others might be considered auxiliary):



- PLGA: Textual representation of PGA.
  Relative jump forward ($\#k$) and instruction sequence repetition ($\backslash\backslash\#k$);
- PLGB: Relocatable programming.
  Relative jump forward ($\#k$) and backward ($\backslash\#k$);
- PLGBpc: Relocatable programming with explicit PC.
  Store relative location (`pc=here`$\pm k$) and jump-to-stored-location `##pc`);

- PLGC: Conventional termination.
  No termination instruction (!); jump outside scope signifies termination;
- PLGD: Absolute addressing.
  Absolute jump ($\#\#k$);
- PLGDpc: Absolute addressing with PC.
  Absolute set (`pc`=k);
- PGLE: Abstract addressing with labels.
  Label definition ($\pounds k$) and jump ($\#\#\pounds k$);
- PGLV: Vector labels.
  Label definition ($[\vec{n}]$) and jump ($\#\#[\vec{n}]$);
- PGLDS: Structured programming.
  Blocks ({ and }), unsized relative jumps ($\#>$ and $<\#$) and skip ($*skip$);
- PGLDSpc: Structured programming with limited non-local capabilities.
  Unsized relative set (`pc=here`$>$ and `pc=`$<$`here`);
- PGLVS: Structured and abstract non-structured addressing: a higher programming language;
- PGLI: A higher programming language with method invocation.
  Call ($R\#\#[\vec{n}]$) and return ($\#\#R$);
- PGLIcf: Component format.
  Define multiple (mutually invoking) PGLI components.

In conclusion it is worthwhile to mention that all projections and examples in this article have been checked by specifying them in an environment specifically suited for the creation of executable PGA-style specifications.

References

[BB02] J. A. Bergstra and I. Behtke. Moleculular dynamics. *The Journal of Logic and Algebraic Programming*, 51(2):193—214, 2002.

[BL02] J. A. Bergstra and M. E. Loots. Program algebra for sequential code. *The Journal of Logic and Algebraic Programming*, 51(2):125—156, 2002.

[BP02] J. A. Bergstra and A. Ponse. Combining programs and state machines. *The Journal of Logic and Algebraic Programming*, 51(2):175—192, 2002.

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0301] J.A. Bergstra and P. Walters, *projection semantics for multi-file programs*, Programming Research Group - University of Amsterdam, 2003.

[PRG0201] I. Bethke and P. Walters, *Molecule-oriented Java Programs for Cyclic Sequences*, Programming Research Group - University of Amsterdam, 2002.

The above reports and more are available through the website: www.science.uva.nl/research/prog/

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

www.science.uva.nl/research/prog/