Bachelor Informatica
Universiteit van Amsterdam

Extending the PGA Toolset

What are the possibilities, restrictions and possible solutions to combining several sets of PGA instructions into a single program?

Stephan Schroevers
Universiteit van Amsterdam
sschroev@science.uva.nl

20th June 2006

Supervisors: Inge Bethke (UvA) and Bob Diertens (UvA)

Signed:

Abstract

Program algebra provides a means to reason about programs and programming languages. One such algebra is PGA. It provides a hierarchy of increasingly complex languages that implement a range of programming constructs. These languages can be translated to each other. The PGA Toolset holds a collection of programs that automates this process. This paper introduces two new languages that add multi-file support to the PGA hierarchy. This is done by defining the necessary translation rules. The PGA Toolset is also updated to handle these new languages and translations.

Table of Contents

CHAPTER 1

Introduction

Program algebra is an algebraic approach to sequential, imperative programs and programming. It allows reasoning about programs and programming languages.

One such program algebra is PGA as defined in [BL02]. In its most basic form it defines basic instructions, test instructions, jump instructions and a termination instruction. Using this very small set of instruction types a hierarchy of higher level languages can be defined in terms of more basic (lower level) languages.

The translation of a higher level language to a lower level language is called a *projection*, whereas the translation of a lower level language into a higher level language is called an *embedding*.

PGA follows simple rules and uses simple instructions. The result is a hierarchy of languages that can easily be remembered and understood at the level of individual instructions.

This paper assumes that the reader is familiar with the projections and embeddings introduced in [BL02]. The languages discussed here are all (indirect) embeddings of the languages mentioned in that document.

1.1 Aims of this paper

As research on programming languages is one of the main purposes of program algebra, there is a constant strive to create more realistic embeddings within PGA. In this respect the final goal of creating new PGA languages is to mimic every aspect of the programming constructs of any language.

Most if not all modern programming languages have support for *functions* (or *methods* in the case of object oriented programming), *macros* and multi-file programs. What these techniques have in common is that they allow the reuse of code and significantly reduce the overhead and complexity of the process of programming and the program code itself.

For this reason it is important to be able to model these concepts within PGA. Significant work has been done in this direction as can be read in e.g. [BB02, BW03].

1.1.1 Multi-file programs

This paper aims to extend the support for multi-file programs within PGA. Currently there is one PGA language that offers this feature: PGLIcf, which is introduced in [BW03]. PGLIcf has its strengths and shortcomings.[1]

This paper adds two new languages with support for multi-file programs to the PGA hierarchy. These languages, which are quite similar, have properties that are significantly different from PGLIcf. These new languages and their projections to existing languages will also be added to the PGA Toolset.

The remaining sections of this chapter will introduce the aspects of PGA that are required to understand the design decisions that are made in this paper and during the extension of the PGA Toolset. A study of the construction of PGLIcf is given. The languages PGLEcm and PGLEcmn, onto which the new languages will be projected, are introduced. The last section provides an overview of the relevant aspects of the PGA Toolset.

Chapter 2 introduces the new languages and projections. Chapter 3 describes the design decisions that were made during the implementation of the new languages as part of the PGA

[1] See Section 1.3, especially Section 1.3.5.

Toolset.

1.2 Primitive instruction sets and basic instruction sets

It is important to understand that PGA makes a distinction between instruction sets that are called *primitive* and *basic*. The primitive instruction sets provide the control constructs of the language. PGLA and its hierarchy of (indirect) embeddings are examples of primitive instruction sets (see Appendix A).

A basic instruction is a parameter of a primitive instruction. Each basic instruction returns a boolean value upon which a primitive instruction can react. A basic instruction set defines a language consisting of instructions that are to be used in conjunction with a primitive instruction set. As an example, consider the following code snippet in the language PGLE:

```
L0; +a; ##L1; ##L0; L1; b
```

Here, the instructions are delimited by a semi-colon. The instructions `L0` and `L1` are primitive label catch instructions. `##L0` and `##L1` are primitive goto instructions. The expressions `a` and `b` are basic instructions. The `+a` operation evaluates the result of `a`. Program execution continues at the next instruction if `a` returns *true* or skips the next function if `a` returns *false*.

When running this program the instuction `a` would be executed as long as it returns false. Once it returns true `b` is executed, after which the program terminates.

1.2.1 A basic instruction set: MPP and its extensions

One of the basic instruction sets that has been created for PGA is MPP, short for *Molecular Programming Primitives*. MPP is introduced in [BB02]. In short it provides a way to model program states by making an analogy with a fluid of molecules. Though it is assumed that the reader is familiar with the concepts of atoms, fields and foci as used in MPP, this section will provide a very short introduction to this language.

Consider Figure 1.1, which shows a fluid consisting of a single molecule and a proto-atom named `null`. The molecule consists of two atoms (the black dots) which each have a single focus: `x` and `y`. The atoms can therefore be referenced by the names `x` and `y`. Atom `x` has a field to atom `y` named `a` and atom `y` has a field to atom `x` named `b`. This molecule is the result of the following PGLE.mpp (PGLE with the MPP basic instruction set) program:

```
x=new; y=new; x.+a; y.+b; x.a=y; y.b=x
```

The first two instructions create new atoms and assign their respective foci. The second two instructions add a reflexive field with the given names to each atom. The last two instructions make the fields point to the other atom. The proto-atom `null` is present to allow the removal of foci from atoms by assigning them to `null`.

In short, MPP provides instructions to create atoms, assign foci to atoms, add and remove fields, direct fields, and test for the equality of atoms and the existence of fields.

Several extensions to MPP have been defined. The most notable extension is MPPV, which introduces additional instructions for dealing with values of basic types. These values can be viewed as terminal objects (i.e. they do not support fields) labeled with a literal. Specifically, MPPV supports values of type boolean and integer.

In [Die04] Bob Diertens describes additional basic instruction sets which are built on top of MPPV. First there is HMPPV (*High-level MPPV*) which adds support for strings among some other additional instructions. HMPPV is embedded within MSP or *Molecular Scripting Primitives*, which provides some instructions that can be considered as replacements of large chunks of code. MSPea adds support for code evaluation to MSP. It must be noted that MSPea introduces one instruction that cannot be translated to MSP. The `apply` instruction evaluates a given string as a basic instruction. However, PGLEc and MSP provide no means to turn a string into a focus or field selection.

1.3 Methods and the multi-file paradigm using primitive instruction sets

One way to model method calls and to split up programs in multiple files is described in the electronic report [BW03] by Bergstra and Walters. This report describes the use of an explicit

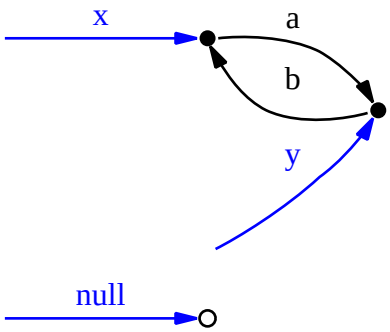

Figure 1.1: A molecule that is the result of an MPP program.

program counter and state machines in order to model method definitions and invocations as well as structured programming within PGA. All languages introduced are indirect embeddings of PGLA and thus primitive instruction sets. This section will give a short overview of the key concepts described in [BW03]. Figure 1.2 places the languages described here in their greater context.

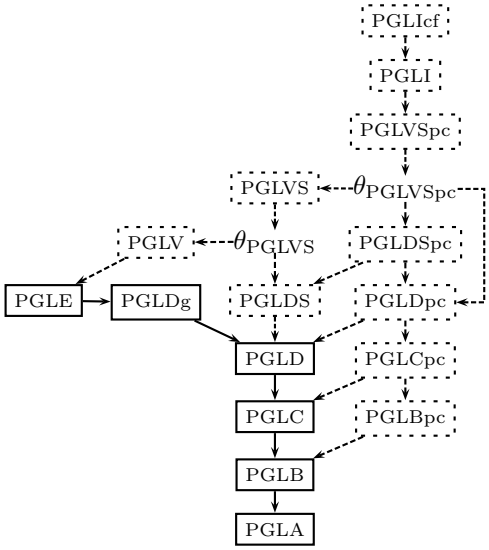

Figure 1.2: Languages and projections introducted in [BW03] (dashed lines) in the context of the languages on which they are based (solid lines).

1.3.1 An explicit program counter

Method calls and other control features within a program require that at times the program counter needs to be manipulated, thereby making it explicit. PGA only knows an implicit program counter. On itself a PGA program cannot retrieve, alter and store this implicit program counter.

To work around this problem, an explicit program counter may be introduced. This program counter will be explicitly altered by the program itself and may be seen as a variable that is modeled using solely primitive instructions. For this purpose, state machines are introduced.

1.3.2 Modeling state machines

[BW03] describes several state machines modeled in PGA. Especially worth noting are the state machines `smpc` which stores a program counter and `smco` which stores a sequence of program counter values, but no duplicate elements.

Though state machines can potentially have an infinite number of different states, the introduced state machines can be modeled using a finite number of PGA instructions because they are only required to store a finite number of different values. To see why, note that every program has a finite number of instructions and that therefore the number of different valid values for the program counter is finite too.

1.3.3 An explicit program counter using a state machine

The concept of an explicit program counter and the state machine `smpc` first meet in the language PGLBpc. PGLBpc provides several instructions that manipulate the program counter by allowing it to be set to a position relative to the calling instruction. PGLBpc can be projected onto PGLB.

The language PGLDpc offers functionality similar to PGLBpc, with the difference that it treats addresses in an absolute instead of a relative manner. PGLDpc can be projected onto PGLBpc using the intermediate language PGLCpc.

Note that both PGLBpc and PGLDpc can only remember the most recent value to which the program counter is set. Therefore, neither of these languages is suitable to simulate, for example, two or more nested method calls.

This problem is partially solved by the language PGLI. PGLI introduces the *returning goto* instruction, allowing the program to jump to some predefined instruction while remembering the location of the goto instruction. When a *return* instruction is executed, the program continues the execution at the instruction following the goto instruction. This involves storing multiple values of the program counter, one for each goto instruction. To achieve this, PGLI makes use of the state machine `smco` instead of `smpc`. As mentioned in the previous section, this state machine has one disadvantage: the stack of which it keeps track may not contain duplicate values. This places a firm restriction on PGLI, in that it cannot handle (mutual) recursion.

1.3.4 Structured programming

[BW03] also introduces structured programming — the concept of code blocks and the ability to either execute or skip them — using the language PGLDS which can directly be projected onto PGLD. As structured programming is not the focus of this paper, PGLDS is not examined in depth. Worth mentioning is that the report also introduces the language PGLDSpc, thereby adding the possibility to manipulate the program counter in PGLDS. This allows exception handling. PGLDSpc is indirectly a component of the language PGLI that was mentioned in the previous section.

1.3.5 Multiple program components

The report introduces the language PGLIcf that allows the use of multiple program components. The suffix 'cf' in the name PGLIcf stands for *component format*. A program component can be regarded as sequence of instructions that is not necessarily capable of acting as a program on its own, but may contain code that is made available to other program components. Program components are the building blocks of source code libraries.

When program components are merged, one may find that some of the components have label catch instructions with identical values in common. Such namespace clashes must be resolved. To do this, vector labels are introduced by means of the language PGLV. PGLI embeds PGLV.

A vector label contains a sequence of integer values that make up a unique label value. This sequence of values can be projected onto a unique single integer using a provided bijective mapping. This ensures compatibility with PGLE, onto which PGLV is projected.

Summarizing, PGLIcf is the first language that supports multiple file components. It is a primitive instruction set and can thus be used in combination with every basic instruction set. Its main drawback is the lack of support for recursion. Furthermore there is currently no embedding of PGLEcf that provides support for parameterized method calls. These two issues are of great importance with respect to code reusability.

1.4 Defining methods using MPP

The article [BB02] examines two interesting concepts with regards to code reuse. The article uses PGLEc.mpp, which means that the basic instruction set MPP is used in conjunction with the primitive instruction set PGLEc.

1.4.1 Molecular programming families

Introduced first is the concept of *molecular programming families*, MPF for short. An MPF can be regarded as a set of programs that are supposed to act on the same molecules or fluids. The programs are (usually small) sequences of instructions that can be embedded in larger programs. Additional restrictions may be imposed on the use of a programming family. For example, parameters are passed through the use of variables with predefined names. The use of variables with such names other than for the purpose of passing arguments should be avoided. Further more, there may also be restrictions on when and where to insert the program components in the main code. An MPF can probably be best compared with a set of macro's, with the important difference that programs in an MPF do *not* substitute variables in a way that macro's do.

An example of an MPF that is introduced in [BB02] is that of the natural numbers. The provided programs are named `setZero` (creates the number 0), `P` (predecessor, decrements a number), `S` (successor, increments a number) and `mod2` (modulo 2). Note that one of the restrictions imposed on this MPF is that `setZero` should always be executed before any of the other programs in the program family.

It is clear that the concept of MPFs becomes impractical when several different programming families are used within the same program. Some of the variable names that they reserve for special purposes may overlap. This results in the need to alter one or more MPFs before they can be used in conjunction with other MPFs. For this reason, methods are a better alternative.

1.4.2 Method calls: PGLEcm and PGLEcmn

[BB02] shows how method definitions and method calls can be added to PGLEc using the MPP instruction set.[2] The language introduced in that paper has no name, so we shall call this embedding PGLEcmn. The reason for this name will become clear shortly. One may pass arguments to the methods defined in PGLEcmn and a return value may be defined. Unlike PGLI, this implementation does support the use of recursion and is thus more flexible. Of course this comes with a price: PGLEcmn cannot be projected onto PGLEc without the use of MPP[3] and is thus not independent of a specific basic instruction set.

The name PGLEcmn is a derivative of the name of the language PGLEcm. PGLEcm is a language implementation in the PGA Toolset that closely resembles the functionality and implementation of method definitions as described in [BB02]. The difference is that the projection semantics in the paper only allow numerical method names, whereas the Toolset implementation supports string names for methods. Hence the appended 'n'. For a precise overview of the differences between these two languages, see Appendix B in which the projection `pglecm2pglecmn` is introduced.

In order to allow the use of methods, the notion of a stack is introduced. Without going into further detail, the stack is represented as a list of atoms with `prev` and `next` fields to each preceding and successive atom. The returning goto instruction, which is also introduced in the language PGLI, manipulates the stack and allows the recursive invocation of methods. The language that introduces the returning goto instruction using MPP is part of the PGA Toolset under the name of PGLEcr. It is to this language that PGLEcm is projected. In turn PGLEcr is projected onto PGLEc.

The arguments that are passed to a method are saved on the stack to preserve their value in the event of recursion (or any other situation in which variables may be overwritten due to overlapping scopes). PGLEcm does not provide a means to discriminate between local and global variables. Hence, all variables may be considered global. This implies that non-argument variables that are used within a method will be overwritten when coping with recursion. The

[2] See pages 206–209.

[3] Or any other sufficiently expressive basic instruction set.

solution for this problem, namely to also add these variables to the stack, is not handled by the projection to PGLEcr and must thus be performed manually.

Note that though both [BB02] and the PGA Toolset use the language MPP to implement method calls, it is very well possible that one may define a completely different basic instruction set which provides enough functionality to define PGLEcr and PGLEcm without the use of MPP.

1.5 Making PGA more user friendly: string labels

Most PGA languages make extensive use of numeric labels. While these greatly simplify the construction of the various projections and embeddings, they have a disadvantage.

In general, the meaning of a numeric label within a program cannot be determined from its value. When programs grow larger, the result may be that the human reader must take great effort to interpret the program. This is not desirable, since PGA was created to be easily readable by humans.

A solution for this problem is the introduction of label values to which clear semantics can be assigned: strings. These string labels can easily be projected onto numeric labels, as is necessary when projecting a language that supports string labels onto a lower level language that does not.

One such projection is described in [Gee03], where string labels are introduced in the language PGLEcws, an embedding of PGLEcw. Each string label is assigned a unique number larger than the largest value of any numeric label present in the program. This is done by ordering all unique strings in program X in a dictionary D^X. For a dictionary of size n, D^X_1 through D^X_n denote all entries and thus all unique strings present in the program. The index of some string s in the dictionary is obtained through the function $D^X(s)$.

The projection of PGLEcws onto PGLEcw, or more generally, the projection of some language supporting numeric and string labels onto some lower language that is identical except for the lack of string label support, is defined below. Here m denotes the maximal numerical label in program X and s is some string present in D^X.

- $\psi(\mathtt{L}s) = \mathtt{L}(D^X(s) + m)$;
- $\psi(\#\#\mathtt{L}s) = \#\#\mathtt{L}(D^X(s) + m)$;
- $\psi(u) = u$, otherwise.

This projection is present in the PGA Toolset in the form of the program *slabel2nlabel*. Note that this implementation rounds the value of m to the next multiple of 1000 in order to make the newly created numerical labels more distinguishable from the old ones.

1.6 The PGA Toolset

The PGA Toolset is an application built for educational and scientific purposes that is meant to serve as a tool while working with the various PGA languages. The Toolset is built and maintained by Bob Diertens as described in [Die03].

1.6.1 Functionality

The simulator provides a command line as well as a graphical user interface. The PGA Toolset can project certain PGA languages onto other languages and has support for several basic instruction sets, such as MPP. See Figure 1.3 for an overview of the primitive instruction sets and projections that are currently implemented in the PGA Toolset.

1.6.2 Program architecture

The Toolset has a modular structure that makes the easily extendable. Because the program is implemented using the programming language Perl (and Tcl/Tk for the graphical views), it can be run on many different platforms. The fact that Perl has excellent support for regular expressions greatly simplifies the parsing of PGA code.

This section will only describe the general structure of the parsers and projection programs in the Toolset. The graphical interface is not relevant for this paper.

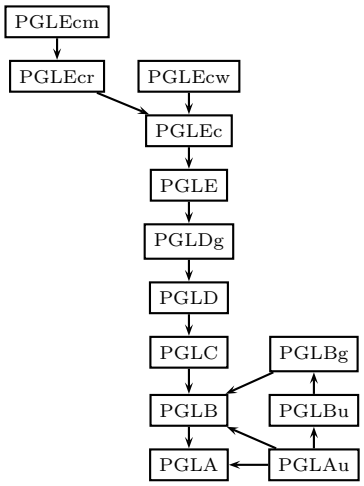

Figure 1.3: Projections that are implemented in the PGA Toolset.

To get an idea of the inner workings of the Toolset, see Figure 1.4. This diagram shows the various modules that are used to project PGA language `b` onto language `a`. The arrows point from a module that is called to the calling module.

Here, the *Input* module reads the program in language *b* and prepares it for parsing. The *Generic* module provides the most basic form of parsing: it strips leading and trailing spaces, but otherwise accepts any token. Because of this it is compatible with every basic instruction set. The *Display* module handles the program's output. This means that none of the other modules has to bother with the differences between the command line and the graphical interface.

The *PrimitiveA* and *PrimitiveB* modules provide functionality to parse code in their respective language and represent this data in an internal, general format (a list of opcodes and arguments). These modules also provide a routine that performs basic checks on the code and a function that translates the internal instruction list to the program representation in its original form.

The program *b2a* uses *PrimitiveB* to parse some program in language `b`, alters the internal program representation according to the projection rules for the projection `b2a` and then uses *PrimitiveA* to print the result in language `a`.

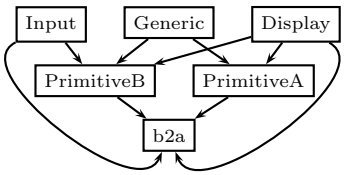

Figure 1.4: Projection model used by the PGA Toolset.

CHAPTER 2

Adding support for multiple files in PGLEcm

This chapter describes two new languages named PGLEcmcf and PGLEcmcfn. They support the use of multiple program files or *program components* and can be projected onto PGLEcm and PGLEcmn, the languages that are described in Section 1.4.2. Specifically, PGLEcmcfn can be projected on PGLEcmn whereas PGLEcmcf can be projected onto PGLEcm as well as PGLEcmcfn.

The choice for PGLEcm as the basis for these new languages is based on the fact that PGLEcm is the only language currently implemented by the Toolset that supports methods. Methods are the basic building blocks within a programming language that allow code reuse and the creation of code libraries. Libraries are collections of methods that provide a specific functionality that may be used by multiple programs. Because libraries are shared among multiple programs, they need to be in separate program files. This is where PGLEcmcf comes in.

As said in Section 1.4.2, PGLEcm and PGLEcmn differ in that the former allows the use of string method names while the latter only allows numerical method names. The only restriction posed on method names defined in PGLEcm is that they consist solely of alphanumeric characters and/or the dash character. PGLEcmn has the advantage that it is easier to project onto PGLEcr, but PGLEcm is clearly more user friendly. Even the code examples in [BB02] use string method names.

2.1 Analysis of techniques used by PGLIcf

Currently there is only one language projected to PGA that allows programs to span multiple files: PGLIcf, an extension of PGLI.[1] It has currently not been added to the PGA Toolset. Of the languages that are part of the PGA Toolset, PGLEcm is probably the language that is functionally the most similar to PGLI, though it can be argued that the properties that PGLEcw shares with PGLI are just as significant. This section analyzes PGLIcf in order to get some clues on how to define the new languages PGLEcmcf and PGLEcmcfn and how to formulate their projection onto PGLEcm(n).

2.1.1 PGLIcf's internals

PGLIcf assumes a single flat file system F in which each file or *component* is identified by a unique integer value. In short, the projection `pglicf2pgli(`c, F`)` of some program c incorporates the program c as well as any program c' that is directly or indirectly referenced by c. Label catch and goto instructions and method declarations and references are resolved using vector labels (see Section 1.3.5). This ensures that the namespaces of the included files will not clash.

As PLGI only regards portions of code in other files of the file system when they are referenced, it has no notion of code that is present outside of the called functions. For this reason the projection `pglicf2pgli(`c, F`)` concatenates all relevant program files by separating them with two termination instructions (`!;!`). To see why two termination instructions are necessary, consider a situation where the last instruction of some included program component is a test instruction. If it were not for the second termination instruction, program flow could jump over

[1] The suffix 'cf' stands for *component format*. PGLI and PGLIcf are introduced in [BW03] and are briefly described in Section 1.3.

the first termination instruction, in which case execution would continue in another unrelated program component. The second termination instruction prevents this and as a result the order in which program components are concatenated after the main component c is irrelevant.

2.2 PGLEcmcfn

This subsection defines the language PGLEcmcfn and its projection onto PGLEcmn, thereby assuming the projection semantics for method calls as described in [BB02].

PGLEcmcfn introduces the notion of multiple program components — sequences of instructions with a name (number) that uniquely identifies them — by defining additional instructions that are prefixed with a component identifier $\texttt{c}j$ using the |-operator. Here j is a natural number referring to some program component c_j in file system F. Method names are of the form $\texttt{m}k()$ or $\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})$ with k a natural number.

The following is a list of the new instructions that PGLEcmcfn provides on top of PGLEcmn. Note that this list is somewhat oriented towards MPP. The |-operator has a higher precedence than the .-operator that is part of MPP.

- $\texttt{c}j|\#\#\texttt{L}k$;
- $\texttt{c}j|\texttt{m}k()$;
- $\texttt{x} = \texttt{c}j|\texttt{m}k()$;
- $\texttt{c}j|\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\texttt{x} = \texttt{c}j|\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\texttt{x.c}j|\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\texttt{y} = \texttt{x.c}j|\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})$.

2.2.1 The projection of PGLEcmcfn onto PGLEcmn

Mimicking `pglicf2pgli(`c, F`)` as closely as possible, the following rules define the projection `pglecmcfn2pglecmn(`c, F`)`. Here, $C = \{c_1, \ldots, c_k\}$, which is the smallest set of program components in file system F that are directly or indirectly referenced by program component c. The integer value q is the largest label or method number present in the program components that are part of C, *plus 1*. By defining q to be larger than any of the label and method numbers present in C, the value 0 can also be used as a label or method number. A program component c_n with instruction sequence p_n is written as $\prec c_n, p_n \succ$.

The function ϕ concatenates all program components in C using the function ψ. ψ appends two termination instructions to each program component after it has been translated by the function θ : PGLEcmcfn $\rightarrow$ PGLEcmn.

- $\phi_C(\prec c_1, p_1 \succ, \ldots, \prec c_k, p_k \succ) = \psi_1^C(p_1); \ldots; \psi_k^C(p_k)$;
- $\psi_n^C(u_1; \ldots; u_m) = \theta_n^C(u_1); \ldots; \theta_n^C(u_m); !; !$;
- $\theta_n^C(u) = !$, when u is a non-local label jump or method invocation (i.e., an instruction $\texttt{c}j|\texttt{m}k(\ldots)$ referring to a module not contained in C. Otherwise,
- $\theta_n^C(\texttt{L}k) = \texttt{L}(nq + k)$;
- $\theta_n^C(\#\#\texttt{L}k) = \#\#\texttt{L}(nq + k)$;
- $\theta_n^C(\texttt{c}j|\#\#\texttt{L}k) = \#\#\texttt{L}(jq + k)$;
- $\theta_n^C(\texttt{m}k()\{) = \texttt{m}(nq + k)()\{$;
- $\theta_n^C(\texttt{m}k()) = \texttt{m}(nq + k)()$;
- $\theta_n^C(\texttt{c}j|\texttt{m}k()) = \texttt{m}(jq + k)()$;

- $\theta_n^C(\texttt{x} = \texttt{m}k()) = \texttt{x} = \texttt{m}(nq + k)()$;
- $\theta_n^C(\texttt{x} = \texttt{c}j|\texttt{m}k()) = \texttt{x} = \texttt{m}(jq + k)()$;
- $\theta_n^C(\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{m}(nq + k)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{c}j|\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{m}(jq + k)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x} = \texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x} = \texttt{m}(nq + k)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x} = \texttt{c}j|\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x} = \texttt{m}(jq + k)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x.m}k(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x.m}(nq + k)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x.c}j|\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x.m}(jq + k)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{y} = \texttt{x.m}k(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{y} = \texttt{x.m}(nq + k)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{y} = \texttt{x.c}j|\texttt{m}k(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{y} = \texttt{x.m}(jq + k)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(u) = u$, otherwise.

2.3 PGLEcmcf: PGLEcmcfn with string names

This section introduces an extension of PGLEcmcfn that allows the use of string labels and string method names. This language, named PGLEcmcf, will be projected onto PGLEcmcfn.

Because the implementation of PGLEcm in the PGA Toolset uses string names too[2], it is relatively straightforward to project PGLEcmcf directly onto PGLEcm. For this reason, such projection is also provided.

PGLEcmcf provides the following instructions instead of those described in Section 2.2. Note that the component identifier c' and the label/method name s are strings.

- $c'|\#\#\texttt{L}s$;
- $c'|s()$;
- $\texttt{x} = c'|s()$;
- $c'|s(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\texttt{x} = c'|s(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\texttt{x}.c'|s(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\texttt{y} = \texttt{x}.c'|s(\texttt{arg1}, \ldots, \texttt{argn})$.

2.4 The projection of PGLEcmcf onto PGLEcmcfn

This section will describe how PGLEcmcf can be translated to PGLEcmcfn. For this purpose a new kind of string dictionary needs to be defined first.

2.4.1 Multiple string dictionaries

The dictionary D^X as introduced in Section 1.5 contains the collection of unique strings in some program X. Specifically, it is used in the projection of PGLEcws onto PGLEcw to convert string labels into numerical labels. PGLEcmcf does not only have string labels, but also string method names and string component names. If the projection `pglecmcf2pglecmcfn`(c, F) were to use the dictionary D^X, where X would be the concatenation of all program components in F, then two things would happen:

- There would be no continuous ordering of label and method names in each program component. The integers would appear to be selected at random. This would *not* improve the readability of the resulting program code.

[2]For methods, not for labels.

- Occurrences of identical strings in different program components would be mapped onto the same integer. This can be regarded as a good thing, considering that it will improve the understandability of the resulting PGLEcmcf code.

It can be argued that the former issue is of greater importance. Therefore, the projection of PGLEcmcf will have a separate dictionary for each program component. Dictionary D^X is thus assumed to contain the unique string names of labels as well as methods in the program component X.

The names of program components that are (indirectly) referenced by program c are not necessarily located in a single program component. The mapping of these names onto integer values will therefore be handled as a special case. The dictionary D^F stores the unique string names of the collection of files in F. The function $D^F(c)$ returns the index of component name c in the dictionary.

Note that because D^X and D^F are dictionaries over different domains, the notation of the functions $D^X(s)$ and $D^F(c)$ is unambiguous.

2.4.2 The projection

The projection `pglecmcf2pglecmcfn(`c, F`)` can now be defined as follows:

- $\phi_C(\prec c_n, p_n \succ) = \psi_1^C(p_n)$;
- $\psi_n^C(u_1; \ldots; u_m) = \theta_n^C(u_1); \ldots; \theta_n^C(u_m)$;
- $\theta_n^C(\texttt{L}s) = \texttt{L}D^{c_n}(s)$;
- $\theta_n^C(\#\#\texttt{L}s) = \#\#\texttt{L}D^{c_n}(s)$;
- $\theta_n^C(c'|\#\#\texttt{L}s) = \texttt{c}D^C(c')|\#\#\texttt{L}D^{c'}(s)$;
- $\theta_n^C(s()\{) = \texttt{m}D^{c_n}(s)()\{$;
- $\theta_n^C(s()) = \texttt{m}D^{c_n}(s)()$;
- $\theta_n^C(c'|s()) = \texttt{c}D^C(c')|\texttt{m}D^{c'}(s)()$;
- $\theta_n^C(\texttt{x} = s()) = \texttt{x} = \texttt{m}D^{c_n}(s)()$;
- $\theta_n^C(\texttt{x} = c'|s()) = \texttt{x} = \texttt{c}D^C(c')|\texttt{m}D^{c'}(s)()$;
- $\theta_n^C(s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{m}D^{c_n}(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(c'|s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{c}D^C(c')|\texttt{m}D^{c'}(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x} = s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x} = \texttt{m}D^{c_n}(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x} = c'|s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x} = \texttt{c}D^C(c')|\texttt{m}D^{c'}(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x}.s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x}.\texttt{m}D^{c_n}(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x}.c'|s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x}.\texttt{c}D^C(c')|\texttt{m}D^{c'}(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{y} = \texttt{x}.s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{y} = \texttt{x}.\texttt{m}D^{c_n}(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{y} = \texttt{x}.c'|s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{y} = \texttt{x}.\texttt{c}D^C(c')|\texttt{m}D^{c'}(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(u) = u$, otherwise.

2.5 The projection of PGLEcmcf onto PGLEcm

PGLEcmcf can also be projected onto PGLEcm instead of PGLEcmcfn, thus preserving string method names instead of converting them to integers. Label names *are* converted to integers. For this reason the projection `pglecmcf2pglecm(`c, F`)` requires two new functions: a special kind of dictionary and a string concatenation function.

2.5.1 String labels and multiple files

The use of string labels described earlier in Section 1.5 assumes a single program component. This is clearly unsuitable for the conversion of multiple program components to a single program file. In order to convert the string labels in PGLEcmcf, an alternative type of dictionary ${D'}^C$ over the collection of program components C is introduced. This dictionary consists of elements ${D'}_1^C$ through ${D'}_n^C$. Each element is a tuple (c, s) that denotes the string s in program component c, where $c \in C$. The index of some tuple (c, s) within the dictionary is returned by the function ${D'}^C(c, s)$.

Now the conversion of label catch and goto instructions in PGLEcmcf to equivalent instructions in PGLEcm becomes:

- $\psi_n^C(\texttt{L}s) = \texttt{L}{D'}^C(c_n, s)$;
- $\psi_n^C(\#\#\texttt{L}s) = \#\#\texttt{L}{D'}^C(c_n, s)$;
- $\psi_n^C(c'|\#\#\texttt{L}s) = \#\#\texttt{L}{D'}^C(c', s)$;
- $\psi_n^C(u) = u$, otherwise.

This alternative approach assumes that there are no numerical labels. Some instruction `L1` in program component $c_n \in C$ will be interpreted as $\psi_n^C(\texttt{L1}) = \texttt{L}{D'}^C(c_n, 1)$, because the numerical value will be regarded as a string. By treating every label value as a string, there is no need to determine the maximum value of the numerical labels over all included program components.

2.5.2 Concatenating strings

In order to translate the method names in PGLEcmcf to those in PGLEcm, they are prefixed with the name of the program component in which they are declared. To do this, the names must be concatenated in such manner that the resulting name cannot be ambiguous in PGLEcm. For this purpose a new function $S(c, s)$ for program component c and method name s is introduced. The function does the following:

- Each occurrence of the dash character (-) within the string c is replaced by two dashes.
- The same operation is performed on the string s.
- The two strings c and s are concatenated with a single dash in between.

As an example, assume the program component `stdio` and the method name `print-it` which is defined within `stdio`. In PGLEcmcf this method will externally be referenced by the name `stdio|print-it`. When projected onto PGLEcm, the name of this method will be translated to `stdio-print--it`.

The function $S(c, s)$ is bijective and does not introduce a new type of character, thereby ensuring that each combination of c and s results in exactly one, unique method name that is valid in PGLEcm.

2.5.3 The projection

The projection `pglecmcf2pglecm(`c, F`)` for some PGLEcmcf program c within file system F becomes:

- $\phi_C(\prec c_1, p_1 \succ, \ldots, \prec c_k, p_k \succ) = \psi_1^C(p_1); \ldots; \psi_k^C(p_k)$;
- $\psi_n^C(u_1; \ldots; u_m) = \theta_n^C(u_1); \ldots; \theta_n^C(u_m); !; !$;
- $\theta_n^C(u) = !$, when u is a non-local label jump or method invocation (i.e., an instruction $\texttt{c}j|\texttt{m}k(\ldots)$ referring to a module not contained in C. Otherwise,
- $\theta_n^C(\texttt{L}s) = \texttt{L}{D'}^C(c_n, s)$;
- $\theta_n^C(\#\#\texttt{L}s) = \#\#\texttt{L}{D'}^C(c_n, s)$;

- $\theta_n^C(c'|\#\#\texttt{L}s) = \#\#\texttt{L}{D'}^C(c', s)$;
- $\theta_n^C(s()\{) = S(c_n, s)()\{$;
- $\theta_n^C(s()) = S(c_n, s)()$;
- $\theta_n^C(c'|s()) = S(c', s)()$;
- $\theta_n^C(\texttt{x} = s()) = \texttt{x} = S(c_n, s)()$;
- $\theta_n^C(\texttt{x} = c'|s()) = \texttt{x} = S(c', s)()$;
- $\theta_n^C(s(\texttt{arg1}, \ldots, \texttt{argn})) = S(c_n, s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(c'|s(\texttt{arg1}, \ldots, \texttt{argn})) = S(c', s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x} = s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x} = S(c_n, s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x} = c'|s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x} = S(c', s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x}.s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x}.S(c_n, s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x}.c'|s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x}.S(c', s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{y} = \texttt{x}.s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{y} = \texttt{x}.S(c_n, s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{y} = \texttt{x}.c'|s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{y} = \texttt{x}.S(c', s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(u) = u$, otherwise.

2.6 PGLEcmcf and PGLEcmcfn programs: additional constraints

As the observant reader will have noticed, the projection of PGLEcmcf(n) onto PGLEcm(n) does not alter basic instructions. The reason for this is that the projection of basic instructions depends on the type of instruction set used. The type of basic instruction set used does not need to be specified in order for the projection of a primitive instruction set to work. In fact, the type of instruction set used may not even be known at this time. An implicit constraint placed on the basic instruction set that is being used is that it must be expressive enough to allow the (indirect) projection of PGLEcm(n) onto PGLEc. If this is not possible, it is not possible to reason about the language at a level lower than PGLEcm. This defeats the purpose of PGA.

Regardless of the type of basic instruction set used, there are several constraints to which the program must adhere in order for it to stay semantically unaltered after the projection of PGLEcmcf(n) onto PGLEcm(n). These constraints may either be manually applied by the programmer or enforced by a set of projection rules that are specific to a basic instruction set.

Here, a set of constraints will be formulated to which a programmer must adhere to ensure that his or her program will not be semantically altered by the projections from PGLEcmcf onto PGLEcm and from PGLEcmcfn onto PGLEcmn.

2.6.1 Global vs. local variables

As PGLEcm and PGLEcmn do not provide a mechanism to declare variables to be either local to a method or global to a program, there is no way that the projection of either PGLEcmcf(n) onto PGLEcm(n) or PGLEcm(n) onto PGLEcr knows if and when to save, restore or remove variables that are local when program control changes its scope.

For this reason, special care must be taken when handling variables. This mostly holds for local variables, but even variables that are global to one program component in PGLEcmcf(n) may not necessarily be global to the entire program after projection onto PGLEcm(n).

Part of these problems can be solved by applying strict naming rules. For example, one may choose to prefix each variable that is local to a method with the name of that method and then prefix each variable that is strictly global to a single program component with the name of that component. Local variables would then be prefixed by the method name as well as the name of a program component. This technique is similar to that used by the projection

`pglecmcf2pglecm`(c, F) with regard to method names. Strictly applying this convention ensures that no two variable names that are supposed to be in different scopes collide.

It is important not to overlook the method arguments in this process. They need to be local, as the programmer will want to ensure that the arguments are not overwritten when another function is being called.

2.6.2 Recursion: direct and indirect

Strict naming conventions do not solve all problems relating to the scope of variables. In the special case of recursion, either direct or indirect[3], local variables may be overwritten.

A possible solution is to explicitly implement a stack data structure onto which local variables are saved before a method is called recursively. When the method returns, the variables are popped from the stack again. This stack must be implemented using the instructions provided by the basic instruction set.

2.6.3 Projections specific to the basic instruction set

Alternatively to the manual bookkeeping of variable scopes, additional projection rules may be defined. These projections are to take care of the translation of basic instruction sets before, during or after the projection of PGLEcmcf(n) onto PGLEcm(n) (or any other two primitive instruction sets for that matter).

Each basic instruction set should therefore provide its own collection of additional rewrite rules that should be considered together with those provided by `pglecmcf2pglecm`(c, F) and `pglecmcfn2pglecmn`(c, F).

Note that the implementations of `pglecm2pglecr` and `pglecmn2pglecr` in the Toolset do in fact translate the methods defined in PGLEcm to more primitive constructs in PGLEcr by explicitly defining a stack in MPP. Argument variables are saved on the stack. For this reason, the Toolset implementation of PGLEcm is restricted to the MPP basic instruction set and its derivatives. Note that this implementation does not push other variables local to a method on the stack when calling another function. This bookkeeping is still left to the programmer.

[3] Also called mutual recursion: two or more functions that recursively call each other.

CHAPTER 3

Implementation of the new projections

During the extension of the PGA Toolset with support for the new languages and projections, the code that handled existing languages and projections proved to be very helpful. Because of the Toolset's modular structure, significant parts of this code could be copied without modification. In fact, the code that describes the new languages introduced here greatly resembles that what has been programmed by Bob Diertens in most aspects.

Because of this it does not seem useful to describe the entire programming process. Instead this chapter describes the key concepts that all newly created files share. Furthermore some of the mayor design decisions that were made during the implementation of the new languages and projections are explained.

3.1 Functionality of a primitive module

Each language that is supported by the Toolset has its own module. Each of these modules provides the following routines:

Init Tells which modules to use for input, parsing of basic instructions and output (display).

Parse Parses the input and saves the instructions in a list of opcodes and corresponding arguments.

PrintStep Returns a formatted string representation of the given opcode and arguments.

Print Formats the instructions that were earlier parsed by the *Parse* routine using the *PrintStep* routine. Indentation is added if requested. The result is sent to the display module that was passed to the *Init* method.

Check Performs basic checks on the previously parsed instructions. For example, this routine tries to find matching opening and closing braces.

3.2 Projection code

The programs in the Toolset that perform a projection of one language onto another are also quite similar in the way they work. In short, the process of projecting `PrimitiveB` onto `PrimitiveA` is as follows:

1. Initialize the modules *PrimitiveA* and *PrimitiveB*.
2. Specify the input stream — usually *stdin*, but not in the case of PGLEcmcf(n).
3. Parse the input stream using the *Parse* routine in module *PrimitiveB*.
4. Perform a check on the parsed data.
5. Iterate over the list of parsed instructions and build a new list of instructions that can be understood by *PrimitiveA*.
6. Output the resulting instruction list using the printing functionality provided by module *PrimitiveA*.

3.3 Representing multiple program components

The languages that are currently known to the Toolset share one property that PGLEcmcf and PGLEcmcfn do not have: they are strictly oriented on single file programs. Each provided program interprets its input as a single, stand alone program.

PGLEcmcf(n) programs may consist of multiple components. Additionally, one component should be selected as the main file, i.e. the component that contains the program's first instructions. The elegant input-output principle using pipes applied by the existing programs in the Toolset is in its current form unsuitable to handle this additional information.

There are several possible implementations that allow the input of multiple program components together with the selection of the main component:

- Each program component in a different file. The introduction of command line arguments.

  While a program can only read a single stream from *stdin*[1], the number of command line options that may be passed to a program is unlimited.

  If each program component is stored in a separate file, their filenames can be communicated by means of command line options. This idea is straightforward to implement.

- Each program component in a different file. The introduction of a meta information file.

  To avoid the use of command line arguments, a new meta information file can be introduced that lists the names of the files that contain the program components. The main program component can also be defined.

- Aggregation of the program components into a single file.

  A way to hold on to the principle of a single input stream, is by defining a file format that allows the description of an undefined number of program components. The format can additionally provide the functionality to define the main component, or the program component first described in the file can be assigned this property.

  The markup of such an aggregation file should be defined in a way that allows no ambiguity between its constructs and the PGA components it separates. A possible solution lies in the use of XML[2]. This implementation would have at least one significant drawback: it is incompatible with one of the main purposes of PGLEcmcf(n): defining code libraries.

- More explicit use of the system's file system.

  Another way to stick with a single direct input stream from the user's point of view, is by assuming that all references to other program components are in fact the names of files resident on the file system. This way each program component is saved in its own file without forcing the user to explicitly pass the names of these files to the Toolset. This increases the ease of use, scales well to large amounts of program components and even allows the use of a special environment variable to list the locations of program components. This is comparable to the use of the system's `$PATH`-variable as it is present on most modern operating systems. This implementation most closely mimics the way that libraries are handled by compilers of most modern programming languages.

It should come as no surprise that during the extension of the Toolset the last option was chosen as the most preferable one. The name of the environment variable that is used to search for program components is `$PGALIB`.

Note that in order to be able to work with multiple input streams, a small patch had to be committed to one of the existing modules in the Toolset. See Appendix C for details.

[1] *Standard input*, the file descriptor of the default input stream of a program.
[2] *Extensible Markup Language*, see `http://www.w3.org/XML/`

CHAPTER 4

Conclusion

This paper has introduced two new languages, added three languages to the PGA Toolset and implemented five projections.[1] By defining additional types of instructions to reference instructions in other program components, the number of instructions defined by PGLEcmcf(n) is significantly larger than the number of instructions defined by PGLEcm(n). These new instructions are semantically only slightly different however. It is because of this that the projection of a multi-file program in PGLEcmcf(n) onto a single file PGLEcm(n) program is straightforward.

This beautifully demonstrates one of the basic principles of PGA: the multi-file paradigm is a powerful concept and yet the hierarchy of PGA languages can easily be extended to incorporate this programming construct without compromising another aspect of the hierarchy in any way.

4.1 Acknowledgments

First of all I would like to thank my supervisors Inge Bethke and Bob Diertens for their support and trust. The weekly conversations were a tremendous help. Secondly I would like to thank my fellow students, especially Anton Bossenbroek and Marc Makkes for the many coffee breaks.

The internet radio station `jungletrain.net` deserves special notice for their drum 'n' bass sounds that got me through the writing of this paper. The rock/metal band TOOL deserves special notice for doing exactly the opposite: their music is a perfect way to instantly slip into another world and forget about the world around me.

[1] The fifth projection is the projection `pglecmn2pglecr`, which is a very minor variation of the projection `pglecm2pglecr` that was written by Bob Diertens. Its specifications are outside the scope of this paper.

Bibliography

[BB02] J.A. Bergstra and I. Bethke. Molecular dynamics. *Journal of Logic and Algebraic Programming*, 51(2):193–214, 2002.

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

[BW03] J.A. Bergstra and P. Walters. Projection semantics for multi-file programs. Electronic report PRG0301, Programming Research Group — University of Amsterdam, May 2003.

[Die03] B. Diertens. A toolset for PGA. Electronic report PRG0302, Programming Research Group — University of Amsterdam, October 2003.

[Die04] B. Diertens. Molecular Scripting Primitives. Electronic report PRG0401, Programming Research Group — University of Amsterdam, June 2004.

[Gee03] R.M. Geerlings. A projection of the object oriented constructs of Ruby to program algebra, November 2003.

APPENDIX A

PGA language overview

Figure A.1 tries to present the reader with an overview of all PGA languages and projections that are mentioned in this paper. The language names surrounded by a solid box are those that are known to the PGA Toolset. The solid arrows show the projections that are available in the Toolset. The language boxes with the long dashed borders are introduced in this paper and added to the Toolset, along with the relevant projections.

The boxes and lines with the short dashed edges show the languages and projections that are introduced by [BW03]. Note that the function θ represents the commutative composition of the connected projections.

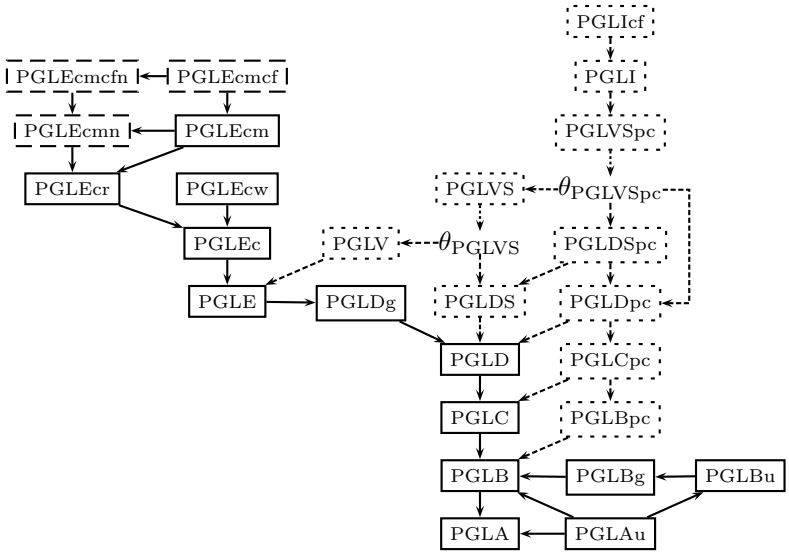

Figure A.1: Projections that are either implemented in the PGA Toolset (solid lines) or mentioned or introduced in this paper (resp. short and loang dashes).

APPENDIX B

The projection of PGLEcm onto PGLEcmn

Throughout this paper a distinction has been made between the specification of the language PGLEcmn as it is introduced in [BB02] and the Toolset implementation in the form of PGLEcm. For the sake of completeness, this section presents the projection `pglecm2pglecmn`. The Toolset has been extended with support for PGLEcmn and the projection of PGLEcm onto this language has also been added.

This projection uses a dictionary D^X very similar to the one introduced in Section 1.5. This version of the dictionary is different in that it also regards method names (not just label names) and in that it disregards the notion of numerical string and method names. This behavior is similar to that of the multi-file dictionary introduced in Section 2.5.1.

These projection rules assume the projection of some program X:

- $\theta_n^C(s()\{) = \texttt{m}D^X(s)()\{$;
- $\theta_n^C(s()) = \texttt{m}D^X(s)()$;
- $\theta_n^C(\texttt{x} = s()) = \texttt{x} = \texttt{m}D^X(s)()$;
- $\theta_n^C(s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{m}D^X(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x} = s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x} = \texttt{m}D^X(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{x}.s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{x}.\texttt{m}D^X(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(\texttt{y} = \texttt{x}.s(\texttt{arg1}, \ldots, \texttt{argn})) = \texttt{y} = \texttt{x}.\texttt{m}D^X(s)(\texttt{arg1}, \ldots, \texttt{argn})$;
- $\theta_n^C(u) = u$, otherwise.

APPENDIX C

The patch of *Input.pm*

Since the new languages PGLEcmcf and PGLEcmcfn allow multiple input streams, the Toolset's *Input* module may be called several times during their projection onto more basic languages; once for each referenced program component. It turned out that a very minor adjustment to this module was necessary in order to be able to call *Input* multiple times during the execution of a single projection. The variable *$myeof* keeps track of the state of the input stream. When *Input* was reinitialized, this variable was not updated. The patch below solves this problem. This patch has been committed to the Toolset.

```
1  --- Input.pm.orig          2006-04-17 15:27:09.588516000 +0200
2  +++ Input.pm      2006-04-17 15:27:26.349920000 +0200
3  @@ -18,6 +18,7 @@
4          my $self = shift;
5
6          $stream = shift;
7  +       $myeof = 0;
8   }
9
10  sub String {
```

APPENDIX D

Short projection example

Figure D.1 aims to provide the reader with a single overview of all the projections described in this paper. The diagram shows a PGLEcmcf program consisting of two program components. The program is projected onto PGLEcmn using two different projection trajectories: via PGLEcmcfn and via PGLEcm.

Note that the label and method numbers in the PGLEcmn version reflect the output of the successive projection of `pglecmcf2pglecmcfn(`c`, `F`)` and `pglecmcfn2pglecmn(`c`, `F`)`. The numbering would have been slightly different if the projections `pglecmcf2pglecm(`c`, `F`)` and `pglecm2pglecmn` would have been used.

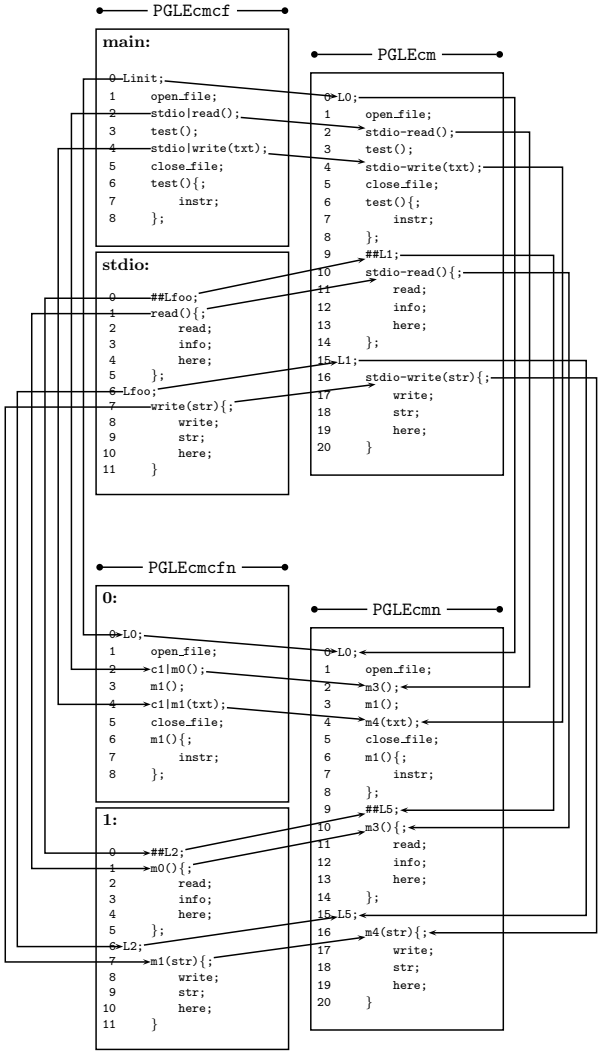

Figure D.1: Example projection of PGLEcmcf onto PGLEcm.

APPENDIX E

Longer projection example

Th following sections provide four versions of a longer and more realistic program than the one shown in Appendix D. The PGLEcmcf version of the program consist of two files. The used basic instruction set is MSP, a language that is introduced in [Die04] and shortly addressed in Section 1.2.1.

The first file, named *geom*, is the main program component. It provides two operations on a geometrical object, namely a cuboid *c*. The second file, *math*, provides the three basic mathematical operations of addition, subtraction and multiplication. This file can be regarded as a very small library. One of the methods the *math* library provides is called by *geom*.

Note that the program uses labels to handle exceptions. The return values of the methods are assigned the focus `that`. The focus `result` points to a boolean literal that contains the value *true* if a method returned normally and *false* if an exception occurred.

E.1 PGLEcmcf.msp

geom

```
    cuboid-volume(c){;
        result = is-cuboid(c);
        + result == false {;
            ##Lcuboid-volume-end;
        };
        that = math|mul(c.w, c.h);
        that = math|mul(that, c.d);
Lcuboid-volume-end;
    };
    cuboid-surface(c){;
        result = is-cuboid(c);
        + result == false {;
            ##Lcuboid-surface-end;
        };
        tmp1 = math|mul(c.w, c.h);
        tmp2 = math|mul(c.w, c.d);
        that = math|mul(c.h, c.d);
        incr that tmp1;
        incr that tmp2;
        that = math|mul(that, 2);
Lcuboid-surface-end;
    };
    is-cuboid(c){;
        that = false;
        - c? {;
            ##Lis-cuboid-end;
        };
        - c/w {;
            ##Lis-cuboid-end;
        };
        - c/w?int {;
            ##Lis-cuboid-end;
        };
        - c/h {;
            ##Lis-cuboid-end;
        };
        - c/h?int {;
            ##Lis-cuboid-end;
        };
        - c/d {;
            ##Lis-cuboid-end;
        };
        - c/d?int {;
            ##Lis-cuboid-end;
        };
        that = true;
Lis-cuboid-end;
    }
```

math

```
    add(n1, n2){;
        result = false;
        that = n1;
        + incr that n2 {;
            result = true;
        };
    };
    sub(n1, n2){;
        result = false;
        that = n1;
        + decr that n2 {;
            result = true;
        };
    };
    mul(n1, n2){;
        result = false;
        that = 0;
        - n1 == 0 {;
            - decr n1 {;
                ##Lmul-end;
            };
            that = mul(n1, n2);
            - incr that n2 {;
                ##Lmul-end;
            };
        };
        result = true;
Lmul-end;
    }
```

E.2 PGLEcmcfn.msp

One may notice that the labels and method numbers in *math* are not in the order of their first occurrence. This is because the file *geom* is parsed first by the Toolset. Thus the method `mul`, which is referenced in *geom*, is assigned a number before the remaining methods in *math* will be parsed.

0

```
    m0(c){;
        result = m1(c);
        + result == false {;
            ##L2;
        };
        that = c1|m0(c.w, c.h);
        that = c1|m0(that, c.d);
L2;
    };
    m3(c){;
        result = m1(c);
        + result == false {;
            ##L4;
        };
        tmp1 = c1|m0(c.w, c.h);
        tmp2 = c1|m0(c.w, c.d);
        that = c1|m0(c.h, c.d);
        incr that tmp1;
        incr that tmp2;
        that = c1|m0(that, 2);
L4;
    };
    m1(c){;
        that = false;
        - c? {;
            ##L5;
        };
        - c/w {;
            ##L5;
        };
        - c/w?int {;
            ##L5;
        };
        - c/h {;
            ##L5;
        };
        - c/h?int {;
            ##L5;
        };
        - c/d {;
            ##L5;
        };
        - c/d?int {;
            ##L5;
        };
        that = true;
L5;
    };
```

1

```
    c1|m1(n1, n2){;
        result = false;
        that = n1;
        + incr that n2 {;
            result = true;
        };
    };
    c1|m2(n1, n2){;
        result = false;
        that = n1;
        + decr that n2 {;
            result = true;
        };
    };
    c1|m0(n1, n2){;
        result = false;
        that = 0;
        - n1 == 0 {;
            - decr n1 {;
                ##L1|3;
            };
            that = c1|m0(n1, n2);
            - incr that n2 {;
                ##L1|3;
            };
        };
        result = true;
c1|L3;
    };
```

E.3 PGLEcm.msp

In PGLEcm the file *geom* and *math* are aggregated into a single program component.

geom

```
    cuboid--volume(c){;
        result = is--cuboid(c);
        + result == false {;
            ##L0;
        };
        that = math-mul(c.w, c.h);
        that = math-mul(that, c.d);
L0;
    };
    cuboid--surface(c){;
        result = is--cuboid(c);
        + result == false {;
            ##L1;
        };
        tmp1 = math-mul(c.w, c.h);
        tmp2 = math-mul(c.w, c.d);
        that = math-mul(c.h, c.d);
        incr that tmp1;
        incr that tmp2;
        that = math-mul(that, 2);
L1;
    };
    is--cuboid(c){;
        that = false;
        - c? {;
            ##L2;
        };
        - c/w {;
            ##L2;
        };
        - c/w?int {;
            ##L2;
        };
        - c/h {;
            ##L2;
        };
        - c/h?int {;
            ##L2;
        };
        - c/d {;
            ##L2;
        };
        - c/d?int {;
            ##L2;
        };
        that = true;
L2;
    };
```

geom (cont'd)

```
    math-add(n1, n2){;
        result = false;
        that = n1;
        + incr that n2 {;
            result = true;
        };
    };
    math-sub(n1, n2){;
        result = false;
        that = n1;
        + decr that n2 {;
            result = true;
        };
    };
    math-mul(n1, n2){;
        result = false;
        that = 0;
        - n1 == 0 {;
            - decr n1 {;
                ##L3;
            };
            that = math-mul(n1, n2);
            - incr that n2 {;
                ##L3;
            };
        };
        result = true;
L3;
    }
```

E.4 PGLEcmn.msp

Note that this version of the program was created through the projection of the PGLEcm onto PGLEcmn. If the PGLEcmcfn version were to be projected onto PGLEcmn, the label and method numbers would have been slightly different.

geom

```
    m0(c){;
        result = m1(c);
        + result == false {;
            ##L0;
        };
        that = m2(c.w, c.h);
        that = m2(that, c.d);
L0;
    };
    m3(c){;
        result = m1(c);
        + result == false {;
            ##L1;
        };
        tmp1 = m2(c.w, c.h);
        tmp2 = m2(c.w, c.d);
        that = m2(c.h, c.d);
        incr that tmp1;
        incr that tmp2;
        that = m2(that, 2);
L1;
    };
    m1(c){;
        that = false;
        - c? {;
            ##L2;
        };
        - c/w {;
            ##L2;
        };
        - c/w?int {;
            ##L2;
        };
        - c/h {;
            ##L2;
        };
        - c/h?int {;
            ##L2;
        };
        - c/d {;
            ##L2;
        };
        - c/d?int {;
            ##L2;
        };
        that = true;
L2;
    };
```

geom (cont'd)

```
    m4(n1, n2){;
        result = false;
        that = n1;
        + incr that n2 {;
            result = true;
        };
    };
    m5(n1, n2){;
        result = false;
        that = n1;
        + decr that n2 {;
            result = true;
        };
    };
    m2(n1, n2){;
        result = false;
        that = 0;
        - n1 == 0 {;
            - decr n1 {;
                ##L3;
            };
            that = m2(n1, n2);
            - incr that n2 {;
                ##L3;
            };
        };
        result = true;
L3;
    }
```