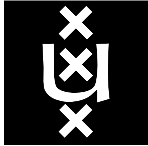

University of Amsterdam

Programming Research Group

A Compiler-projection from PGLEc.MSPio to Parrot

B. Diertens

Report PRG0403 October 2004

B. Diertens

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7593
e-mail: bobd@science.uva.nl

Programming Research Group Electronic Report Series

A Compiler-projection from PGLEc.MSPio to Parrot

Bob Diertens

Programming Research Group, Faculty of Science, University of Amsterdam

ABSTRACT

To gain an insight to projecting a Program Algebra based language to an executable form, we developed a compiler-projection. We used a language with simple control flow and actions based on molecular dynamics as starting point, and a virtual machine as target. To be able to interact with the environment we extended our language with a simple input and output system. In this article, we describe the compiler-projection process.

Keywords: program algebra, code generation

1. Introduction

In program algebra description of language features and mappings to languages with other features are studied. A language is built up from a primitive instruction set (for control flow) with as parameter a basic instruction set (the actions), which return a boolean value upon which a primitive instruction may act. A hierarchy of primitive instruction sets has been developed for which projection (a mapping from an 'higher' to a 'lower' instruction set) and embeddings (a mapping from a 'lower' to an 'higher' instruction set) exist. Sometimes such a mapping is done with the use of a basic instruction set to be able to build the necessary datastructure.

Although a programming language can be used for purely theoretical purposes, i.e. to express a solution to a problem, mostly a user of a programming language wants to execute his programs. The process of converting a program into an executable format, compilation, is nothing more than a mapping from one language to another, in which the latter can be used as the input for an execution system. But with compilation we do not only map the primitive instruction set onto another primitive instruction set, we map both basic and primitive instruction sets onto another combination of basic and primitive instruction sets.
In this article we describe a minimal programming language based on program algebra that is both usable as a language to program in and can easily be turned into an executable format. Furthermore, we describe the process of projecting this language to an executable form. Normally, a projection goes towards a lower notation in our hierarchy of instruction sets, but now we go away from this hierarchy. To distinguish this kind of projection from the others, we call it a compiler-projection.

As our starting point for the programming language we use MSP [8] as basic instruction set and PGLEc as our primitive instruction set. MSP is based on HMPPV, which is a high-level version of MPPV (Molecular Programming Primitives with Values), and extends it with operations on values.
In general, a program acts on some input data and produces some output data. In order to be able to do this we extend MSP with instructions for input and output in a simple form, reading from the front and writing to the end of a sequence of characters (which can be seen as a representation of a file).

As execution system we use Parrot. Parrot is a virtual machine currently being developed for version 6 of Perl [2], but it also aims for being a virtual machine for similar languages, such as Python [4] and Ruby [5], as well.

In the following sections, we give a short overview of program algebra and give descriptions of HMPPV, MSP, and PGLEc.

1.1 Program Algebra

Program Algebra (PGA) [3] is an algebraic framework for sequential programming. It is intended to contribute to a better understanding of sequential programming. A very simple program notation is used as basis for development of other program notations.

The syntax of program expressions in PGA is generated from a set of constants, the primitive instructions, and two composition mechanisms. The primitive instructions have as parameter a set of basic instructions. These basic instructions can be viewed as requests to an environment to provide some service. Upon execution, a basic instructions returns a boolean value.

Primitive instructions:

- a basic instruction: a
  - After performing the basic instruction a execution continues with the next instruction.
- termination: `!`
  - Execution stops.
- positive test: $+a$
  - If the basic instruction a returns true, execution is continued with the next instruction. If it returns false, the next instruction is skipped.
- negative test: $-a$
  - If the basic instruction a returns false, execution is continued with the next instruction. If it returns true, the next instruction is skipped.
- forward jump: $\#k$
  - The instruction itself and the following $k - 1$ instructions are skipped. If k is 0, a jump to the instruction itself is made.

Compositions:

- concatenation of X and Y: X;Y
- repetition of X: X^{ω}

PGLA is a program notation for representing PGA expressions. For dealing with repetition, PGLA has an additional primitive instruction, which replaces repetition.

- repeat: $\backslash\backslash\#n$
  - Here, n is a natural number greater than zero. A program text ending with this instruction will repeat its last n instructions, excluding the repeat instruction itself.

On top of PGLA, other program notations are designed, which can be mapped on PGLA. Such a mapping towards PGLA is called a projection. Several projections to intermediate languages may be used to get the result that is needed. A mapping away from PGLA is called an embedding.

1.2 Molecular dynamics

In molecular dynamics [6] the memory state of a system is modeled as a fluid consisting of a collection of atoms which may have bindings between them to form molecules. A molecule consists of a number of atoms all reachable from one of the atoms by sequences of directed links. A directed link from one atom to another atom exists if the former has a so-called *field* containing the latter. By means of actions causing a change of state, fields can be added to and withdrawn from atoms, and contents of fields can be modified. Selected atoms can also be brought into *focus*.
MPP (Molecular Programming Primitives) is a basic instruction set based on this setting. There is also a version with values (MPPV).

1.3 HMPPV

HMPPV is a basic instruction set based on MPPV (Molecular Programming Primitives with Values). It has some instructions that are a shorthand for commonly used combinations of instructions in MPPV. Furthermore, in addition to the type of values provided by MPPV, it has strings.

With the instructions described here, *extfocus* (extended focus) is used to denote either a focus, or a field selection that may also be compound. When more than one *extfocus* is used in an instruction, they are followed by a number. A field selection that does not exist results in a failure of the instruction. Where not mentioned, an instruction returns `true`, except when a failure occurs, in which case it returns `false`. To make the description of the instructions easier, we consider an atom a value of type atom.

extfocus `= new`
: Create a new atom and assign it to *extfocus*, When *extfocus* is a field selection, it must be of type atom.

extfocus1 `=` *extfocus2*
: Assign the value in *extfocus2* to *extfocus1*. When *extfocus1* is a field selection, the field must be of the same type as the value in *extfocus2*.

extfocus`.+`*f*
: Add a field of type atom to the atom selected by *extfocus*.

extfocus`.+`*f* `= new`
: Add a field of type atom to the atom selected by *extfocus*, and assign it a newly created atom.

extfocus1`.+`*f* `=` *extfocus2*
: Add a field of type atom to the atom selected by *extfocus1*, and assign it the atom selected by *extfocus2*.

extfocus`.+`*f*`:`*t*
: Add a field of type *t* to the atom selected by *extfocus*.

extfocus`.+`*f*`:`*t* `=` *u*
: Add a field of type *t* to the atom selected by *extfocus*, and assign it the value *u*.

extfocus1`.+`*f*`:`*t* `=` *extfocus2*
: Add a field of type *t* to the atom selected by *extfocus1*, and assign it the value selected by *extfocus2*.

extfocus`.-`*f*
: Removes the field *f* from the atom selected by *extfocus*. Returns `false` when the atom has no field *f*.

extfocus`/`*f*
: Returns `true` when the atom selected by *extfocus* has a field *f*, and `false` otherwise.

extfocus1 `==` *extfocus2*
: Comparison of the values selected by *extfocus1* and *extfocus2*. Returns `true` when the values selected by *extfocus1* and *extfocus2* are the same or contain the same atom, and `false` otherwise.

extfocus`?`
: Returns `true` when the value selected by *extfocus* is an atom, and `false` otherwise.

extfocus`?`*t*
: Returns `true` when the value selected by *extfocus* is of the type *t*, and `false` otherwise.

The type of a value field can be one of bool, int, or str, representing boolean, integer, or string. Boolean values can either be `true` or `false`. An integer must be a non-negative number. In notation, string values must be delimited by double quotes ("), and a backslash (\) and a double quote inside a string must be preceded (escaped) by an additional backslash. For the moment, we restrict the set of characters that may appear in a string to the alphanumeric, the punctuation characters, the space, and the newline in the form of \n.

1.4 MSP

MSP extends HMPPV with operations on values. Here, we only describe the instructions that are part of the extension.

With the instructions described here, an `extfocus` is used to denote either a focus, or a field selection that may also be compound. A field selection that does not exist results in a failure of the instruction. A value taken from a focus and all field selections must be of the proper type, otherwise the instructions fails.

Where not mentioned, the instructions return `true`, except when a failure occurs, in which case it returns `false`.

Operations on numbers:

`incr` *`extfocus`*
: Increments the integer value selected by *`extfocus`* with `1`.

`incr` *`extfocus n`*
: Increments the integer value selected by *`extfocus`* with *`n`*.

`incr` *`extfocus1 extfocus2`*
: Increments the integer value selected by *`extfocus1`* with the integer value selected by *`extfocus2`*.

`decr` *`extfocus`*
: Decrements the integer value selected by *`extfocus`* with `1`. Return `false` if the value was already `0`.

`decr` *`extfocus n`*
: Decrements the integer value selected by *`extfocus`* with *`n`* if the value is larger than or equal to `n`. Returns `false` otherwise.

`decr` *`extfocus1 extfocus2`*
: Decrements the integer value selected by *`extfocus1`* with the integer value selected by *`extfocus2`* if it is larger than or equal to the value it must be decremented with. Returns `false` otherwise.

Operations on strings:

`first` *`extfocus1 extfocus2`*
: Takes the first character of the string selected by *`extfocus1`* and assign it as a string of length 1 to *`extfocus2`*. If the selected string is the empty string, `false` is returned.

`delfirst` *`extfocus`*
: Removes the first character from the string selected by *`extfocus`*. If the string is the empty, `false` is returned.

`append` *`extfocus1 extfocus2`*
: Appends the string selected by *`extfocus2`* to the end of the string selected by *`extfocus1`*.

Type conversions:

`int` *`extfocus1 extfocus2`*
: Converts the string selected by *`extfocus1`* to an integer value and assigns it to *`extfocus2`*. If no integer value can be obtained from the string, `false` is returned.

`str` *`extfocus1 extfocus2`*
: Converts the integer value selected by *`extfocus1`* to a string and assigns it to *`extfocus2`*.

1.5 PGLEc

The primitive instruction set PGLEc is the same as PGLE but extended with conditional constructs. We give here a listing of all instructions in PGLEc.

`a`
: The basic instruction *`a`*.

`!`
: Indicates termination of the program.

`+` *`a`*
: When the basic instruction *`a`* returns true, execution continues with the next instructions. When it returns false, the next instruction is skipped.

`-` *`a`*
: When the basic instruction *`a`* returns false, execution continues with the next instructions. When

it returns true, the next instruction is skipped.

`L`*k*

Denotes a label. Here, *k* is a natural number. As an action, it is skipped.

`##L`*k*

Jumps to the first label with the number *k*.

`+` *a* `{`

When the basic instruction *a* returns true, execution continues with the instruction after the matching `}{`, or if there is no matching `}{`, the matching `}`. When it returns false, the next instruction is skipped.

`-` *a* `{`

When the basic instruction *a* returns false, execution continues with the instruction after the matching `}{`, or if there is no matching `}{`, the matching `}`. When it returns true, the next instruction is skipped.

`}{`

Execution continues with the instruction after the matching `}`.

`}`

Execution continues with the following instruction.

2. MSP with I/O

So far, our PGA languages lack any kind of input and output mechanism, and so cannot interact with the environment. Here, we extend MSP with three instructions that enables us to abstract from interacting with the environment, but are powerful enough to mimic a simple input and output mechanism.

`read` *extfocus*

Deletes the first character (actually a string with length 1) from the string in focus `INPUT` and puts it in *extfocus*.

`write "string"`

Appends "string" to the string in focus `OUTPUT`.

`write` *extfocus*

Appends the string in *extfocus* to the string in focus `OUTPUT`.

The `read` instruction is equivalent with the MSP instructions

```
first INPUT extfocus;
delfirst extfocus
```

but with the use of the boolean value returned by `first` in conditional context. The `write` instructions are equivalent with the instructions `append OUTPUT "string"` and `append OUTPUT` *extfocus* .

3. Parrot

We give here a very short description of Parrot. More information can be found in [10] and the documentation that comes along with the source code,[1] which is available from [1].

The Parrot Virtual machine is a register-based virtual machine, which means that the operands for an operation are stored in registers. Internally it uses the Parrot Byte Code format (PBC). Since this is a binary format, another form is used to program in. This format called Parrot Assembler (PASM) has many low-level features, but since it is designed to implement dynamic high-level languages, it also has support for many advanced features.

3.1 Data Types

Parrot has four basic data types, integer, floating-point (also called number), string and Parrot Magic Cookie, or PMC. A PMC is a common base variable type. Each PMC contains some data and has a table

---

1. We used version 0.1.0 with an additional patch to get the macro's working. This patch was made by us and is incorparated in the current development version.

of function pointers, vtable, attached to it. The data can represent an integer, floating-point, string, or a pointer to other data. The list of functions and the place in the vtable is the same for each PMC, and provides the interface to the data type. PMC's can be used to define one owns types, but there are also a number of built-in types.

An aggregate PMC is a PMC that allows indexed access to a sub-element that it stores or references by means of the functions in the attached vtable. An index can be any of the basic data types.

3.2 Registers

Parrot has four sets of 32 registers, one for basic type indicated with the first letter of the basic type (I, N, S, P) and a number from 0 until 31.

3.3 Garbage collection

Parrot has two separate allocation systems builtin into it, each with its own garbage collection scheme. One system is responsible for PMC and string structures, which are of fixed size. The other system is responsible for the contents of strings and PMCs. PASM has instructions for disabling, re-enabling, and forcing garbage collection.

3.4 PIR

PIR stands for Parrot's Intermediate Representation[2] and IMCC is a compiler for PIR that is integrated into Parrot. PIR compiles directly to PBC or to PASM and is the preferred target language for compilers for the Parrot Virtual Machine.

PIR is a medium level language that abstracts from the PASM and allows unlimited symbolic registers. The PIR compiler has a builtin register allocator and spiller.

3.4.1 PIR variables

PIR has two classes of variables, symbolic registers and named variables. Both are mapped to real registers. Named variables must be declared and can either be global or local. Symbolic registers have a $ sign for the first character followed by the character indicating the type of register (I, N, S, P) and one or more digits.

3.5 Example

Below we give an example in which (a) is a Perl program, (b) an equivalent program in PIR, and (c) the translation of (b) into PASM done by the compiler.

```
$i = 6;
$j = 3;
$k = 4;
print $i * ($j + $k);
```

(a)

```
.sub _main
    .local int i
    .local int j
    .local int k
    i = 6
    j = 3
    k = 4
    $I0 = j + k
    $I0 = $I0 * i
    print $I0
    end
.end
```

(b)

```
_main:
        set I18, 6
        set I17, 3
        set I16, 4
        add I16, I17, I16
        mul I16, I16, I18
        print I16
        end
```

(c)

---

2. The original term was PIR, but later on IMC (Intermedicate Code) was used. Now , it seems that PIR is the preferred term.

Note the use of register `I16` for both the named variable `k` and the symbolic register `$I0`.

4. Projecting PGLEc.MSPio to PIR

4.1 Execution Model

In program algebra, a basic instruction returns a boolean value on which the primitive instruction containing the basic instruction can act. Parrot has only the `if` (and the `unless`) instructions that can perform a goto depending on the result of the evaluation of an expression. Somehow, we have to map the program algebra model to the Parrot model.

In general, a basic instruction is compiled into a block of Parrot-code. At certain points in this block the execution of the block is ended. At such points, the result must be stored somewhere and a jump to the end of the block must be made.
At the end of the block the stored result can be tested, in case of a conditional primitive instruction, and acted upon. So the compilation of `+ a {;...;}{;...;}` may look as follows (the register $I31 is used for the storage of the result).

```
        ⋮
    $I31 = 1
    goto ENDBLOCK
        ⋮
    $I31 = 0
    goto ENDBLOCK
        ⋮
    $I31 = 1
ENDBLOCK:
    if $I31 == 0 goto ELSE
        ⋮
    goto ENDIF
ELSE:
        ⋮
ENDIF:
```

We can optimize this scheme by not storing the result and directly jump to `ELSE` if the result is false (0). Then it will look like the following.

```
        ⋮
    goto ENDBLOCK
        ⋮
    goto ELSE
        ⋮
ENDBLOCK:
        ⋮
    goto ENDIF
ELSE:
        ⋮
ENDIF:
```

We can generalize this, by introducing a code generation function ϕ, that takes a basic instruction as its first argument for which it has to generate code, and a label as its second argument to which to jump to if the result is false. Then our scheme looks as follows.

```
φ(a, ELSE)
        ⋮
    goto ENDIF
ELSE:
        ⋮
```

```
ENDIF:
```

Applying this scheme to `- a {;...;}{;...;}` may give the result below.

```
φ(a, IF)
    goto ELSE
IF:
      ⋮
    goto ENDIF
ELSE:
      ⋮
ENDIF:
```

4.2 Data Representation

To represent our fluid we make use of the builtin type `PerlHash`. We use the following code to define the foci.

```
.local PerlHash focus
```

We use a PMC of type Perlhash for every object in the fluid, whether it is an atom or a value. A field is represented by an index of an PMC. The fields `"_t"` and `"_v"` are special (normal fields may not begin with an underscore) and represent the type and value of the object. So, with the following code we add the focus null to our fluid for which we also add a named variable.

```
$P0 = new PerlHash
$P0["_t"] = "atom"
focus["null"] = $P0
.local PerlHash null
null = focus["null"]
```

For example, to following code can be generated for the instruction `x = new`.

```
$P0 = new PerlHash
$P0["_t"] = "atom"
focus["x"] = $P0
```

And the following for `x.+f:int = 42`.

```
$P0 = focus["x"]
$P1 = new PerlHash
$P1["_t"] = "int"
$P1["_v"] = 42
$P0["f"] = $P1
```

This is without code for checking if focus `x` exists, and if it hasn't already a field `f`.

For the input and output system we introduce the named variables `input` and `output` and initialize them with the following code.

```
.local pmc input
.local pmc output
getstdin input
getstdout output
```

4.3 Projection of PGLEc.MSPio to PIR

First, we provide an intermediate program notation PGLEca in which an instruction containing an opening brace is annotated with the number of the instruction containing the matching closing brace. The projection `pglec2pgleca` can easily be done by searching ahead in the program for a matching closing brace.[3] Our projection of PGLEca.MSPio to PIR start with providing every instruction with a label consisting of 'IL' and its instruction number.

pgleca-mspio2pir(u_0 , u_1 , . . . , u_k) =
initialization code
IL0: ψ_0 (u_0)
IL1: ψ_1 (u_1)
$\vdots$
ILk: ψ_k (u_k)
IL$k+1$:
END:
end code

The *initialization code* consist of the head of a function and initialization of our data-types, end the *end code* of the ending of the function.

The auxiliary operations ψ_i are determined by the following rewrite rules:

ψ_i (!) =	goto END
ψ_i (Lk) =	LABELk:
ψ_i (##Lk) =	goto LABELk
ψ_i ($+a$) =	ϕ(a , IL$i+2$)
ψ_i ($-a$) =	ϕ(a , IL$i+1$) goto IL$i+2$
ψ_i ($+a\{n$) =	ϕ(a , ILn)
ψ_i ($-a\{n$) =	ϕ(a , IL$i+1$) goto ILn
ψ_i ($\}\{n$) =	goto ILn
ψ_i ($\}$) =	
ψ_i (a) =	ϕ(a , IL$i+1$)

The second argument of the auxiliary operator ϕ is the label to which a jump is made when the execution of the instruction a may not be continued anymore, as discussed in section 4.1 on the execution model.

The operation ϕ for the MSPio instructions is determined by the rewrite rules below. Here, the registers and labels used in a rewrite rule are local to that rule.

ϕ(incr x , *false*) =
LValue($P0 , x , int , *false*)
```
    $I0 = $P0["_v"]
    inc $I0
    $P0["_v"] = $I0
```

ϕ(incr x v , *false*) =
LValue($P0 , x , int , *false*)
```
    $I0 = $P0["_v"]
    $I0 = $I0 + v
    $P0["_v"] = $I0
```

ϕ(incr x y , *false*) =
ExistingLValue($P1 , y , int , *false*)
LValue($P0 , x , int , *false*)
```
    $I0 = $P0["_v"]
    $I1 = $P1["_v"]
    $I0 = $I0 + $I1
    $P0["_v"] = $I0
```

---

3. A similar thing is done in [7]. There, the instructions containing closing braces are annotated with the number of the instruction containing the matching opening brace.

```
φ(decr x, false) =        LValue($P0, x, int, false)
                              $I0 = $P0["_v"]
                              if $I0 == 0 goto false
                              dec $I0
                              $P0["_v"] = $I0

φ(decr x v, false) =      LValue($P0, x, int, false)
                              $I0 = $P0["_v"]
                              if $I0 < v goto false
                              $I0 = $I0 - v
                              $P0["_v"] = $I0

φ(decr x y, false) =      ExistingLValue($P1, y, int, false)
                          LValue($P0, x, int, false)
                              $I0 = $P0["_v"]
                              $I1 = $P1["_v"]
                              if $I0 < $I1 goto false
                              $I0 = $I0 - $I1
                              $P0["_v"] = $I0

φ(first x y, false) =     ExistingLValue($P0, x, str, false)
                              $S0 = $P0["_v"]
                              length $I0, $S0
                              if $I0 == 0 goto false
                          LValue($P1, y, str, false)
                              substr $S1, $S0, 0, 1
                              $P1["_v"] = $S1

φ(delfirst x, false) =    ExistingLValue($P0, x, str, false)
                              $S0 = $P0["_v"]
                              length $I0, $S0
                              if $I0 == 0 goto false
                              dec $I0
                              substr $S0, $S0, 1, $I0
                              $P0["_v"] = $S0

φ(int x y, false) =       ExistingLValue($P0, x, str, false)
                              $S0 = $P0["_v"]
                              $I0 = $S0
                              abs $I0
                              $S1 = $I0
                              if $S1 != $S0 goto false
                          LValue($P1, y, str, false)
                              $P1["_v"] = $I0

φ(str x y, false) =       ExistingLValue($P0, x, int, false)
                          LValue($P1, y, int, false)
                              $S0 = $P0["_v"]
                              $P1["_v"] = $S0

φ(append x v, false) =    ExistingLValue($P0, x, str, false)
                              $S0 = $P0["_v"]
                              concat $S0, v
                              $P0["_v"] = $S0
```

```
φ(append x y, false) =    ExistingLValue($P0, x, str, false)
                          ExistingLValue($P1, y, str, false)
                              $S0 = $P0["_v"]
                              $S1 = $P1["_v"]
                              concat $S0, $S1
                              $P0["_v"] = $S0

φ(read x, false) =        LValue($P0, x, str, false)
                              read $S0, input, 1
                              if $S0 == "" goto false
                              $P0["_v"] = $S0

φ(write x, false) =       ExistingLValue($P0, x, str, false)
                              $S0 = $P0["_v"]
                              print output, $S0

φ(write v, false) =           print output, PrintValue(v)
```

We continue with the rewrite rules of the operator ϕ for the HMPPV instructions.

```
φ(focus = new, false) =   New($P0, atom)
                              focus["focus"] = $P0
φ(x = new, false) =       LValue($P0, x, atom, false)
                          New($P1, atom)
                              $P0["_v"] = $P1

φ(focus = null, false) =      $I0 = defined focus["focus"]
                              unless $I0 goto false
                              delete focus["focus"]

φ(x.+f, false) =          ExistingLValue($P0, x, atom, false)
                          NoField($P0, f, false)
                              $P0["f"] = $P0

φ(x.+f:type, false) =     ExistingLValue($P0, x, atom, false)
                          NoField($P0, f, false)
                          New($P1, type)
                              $P1["_v"] = DefaultValue(type)
                              $P0["f"] = $P1

φ(x.+f = new, false) =    ExistingLValue($P0, x, atom, false)
                          NoField($P0, f, false)
                          New($P1, atom)
                              $P0["f"] = $P1
φ(x.+f = y, false) =      ExistingLValue($P0, x, atom, false)
                          NoField($P0, f, false)
                          ExistingLValue($P1, y, atom, false)
                              $P0["f"] = $P1

φ(x.+f:type = v, false) =
                          ExistingValue($P0, x, atom, false)
                          NoField($P0, f, false)
                          New($P1, type)
                          AssignValue($P1, type, v)
                              $P0["f"] = $P1
```

```
φ( x.+f:type = y, false ) =
                        ExistingLValue( $P1 , y, type , false )
                        ExistingLValue( $P0 , x, atom , false )
                        NoField( $P0 , f , false )
                        New( $P2, type )
                        AssignPValue( $P2 , type , $P1 )
                            $P0["f"] = $P2

φ( x.-f , false ) =     ExistingLValue( $P0 , x, atom , false )
                        HasField( $P0 , f , false )
                            delete $P0["f"]

φ( focus = y, false ) = Value( $P1 , y, false )
                            $S1 = $P1["_t"]
                        SetFocus( focus, $P1 , $S1 , false )
φ( x.f = y, false ) =   Value( $P1 , y, false )
                        ExistingLValue( $P0 , x, atom , false )
                        HasField( $P0 , f , false )
                            $S1 = $P1["_t"]
                            if $S1 != "atom" goto LL − ELSE
                        SetField( $P0 , f , $P1 , false )
                            goto LL − END
                        LL − ELSE:
                        SetFieldValue( $P0 , f , $P1 , $S1 , false )
                        LL − END:

φ( x = v, false ) =     LValue( $P0 , x, Type( v ) , false )
                        AssignValue( $P0 , Type( v ) , v )

φ( x == y, false ) =    Value( $P0 , x, false )
                        Value( $P1 , y, false )
                        ComparePValue( $P0 , $P1 , false )

φ( x == v, false ) =    ExistingLValue( $P0 , x, Type( v ) , false )
                        CompareValue( $P0 , Type( v ) , v, false )

φ( x/f , false ) =      ExistingLValue( $P0 , x, atom , false )
                            $I0 = defined $P0["f"]
                            unless $I0 goto false

φ( x? , false ) =       ExistingLValue( $P0 , x, atom , false )
φ( x?type , false ) =   ExistingLValue( $P0 , x, type , false )
```

The rewrite rules for the auxiliary functions used in the rewrite rules of operator ϕ are given below.

```
Lvalue( Px , focus, type , false ) =
                        Focus( Px , focus, LL − ELSE )
                        IsType( Px , type , LL − ELSE )
                            goto LL − END
                        LL − ELSE:
                        New( Px , type )
                            focus["focus"] = Px
                        LL − END:
```

```
LValue ( Px , fieldselection , type , false ) =
                          SelectField ( Px , fieldselection , false )
                          IsType ( Px , type , false )
                              eq_addr Px , null , false

ExistingLvalue ( Px , focus , type , false ) =
                          Focus ( Px , focus , false )
                          IsType ( Px , type , false )

ExistingLValue ( Px , fieldselection , type , false ) =
                          LValue ( Px , fieldselection , type , false )

Focus ( Px , focus , false ) =    $I0 = defined focus[ "focus" ]
                              unless $I0 goto LL − ELSE
                              Px = focus[ "focus" ]
                              goto LL − END
                          LL − ELSE :
                              Px = focus["null"]
                          LL − END :

IsType ( Px , type , false ) =    $S0 = Px[ "_t" ]
                              if $S0 != "type" goto false

New ( Px , type ) =               Px = new PerlHash
                              Px[ "_t" ] = "type"

SelectField ( Px , x. fields , false ) =
                          Focus ( Px , x , false )
                          Field ( Px , fields , false )

Field ( Px , f , false ) =        IsType ( Px , atom , false )
                          HasField ( Px , f , false )
                              Px = Px[ "f" ]
Field ( Px , f . fields , false ) =
                          Field ( Px , f , false )
                          Field ( Px , fields , false )

HasField ( Px , f , false ) =     $I0 = defined Px[ "f" ]
                              unless $I0 goto false

NoField ( Px , f , false ) =      $I0 = defined Px[ "f" ]
                              if $I0 goto false

AssignValue ( Px , int , v ) =
                              Px[ "_v" ] = v
AssignValue ( Px , bool , true ) =
                              Px[ "_v" ] = 1
AssignValue ( Px , bool , false ) =
                              Px[ "_v" ] = 0
AssignValue ( Px , str , v ) =
                              Px[ "_v" ] = PrintValue ( v )
```

```
AssignPValue( Px, int|bool, Py) =
            $I0 = Py["_v"]
            Px["_v"] = $I0
AssignPValue( Px, str, Py) =
            $S0 = Py["_v"]
            Px["_v"] = $S0

SetFocus( focus, Px, Sx, false) =
            if Sx != "atom" goto LL-ELSE1
            focus["focus"] = Px
            goto LL-END1
        LL-ELSE1:
            $I0 = defined focus["focus"]
            unless $I0 goto LL-ELSE2
            $P0 = focus["focus"]
            $S0 = $P0["_t"]
            if $S0 == "atom" goto LL-ELSE2
            goto LL-END2
        LL-ELSE2:
            $P0 = new PerlHash
            focus["focus"] = $P0
        LL-END2:
            $P0["_t"] = Sx
            if Sx != "str" goto LL-ELSE3
            $S0 = Px["_v"]
            $P0["_v"] = $S0
            goto LL-END3
        LL-ELSE3:
            $I0 = Px["_v"]
            $P0["_v"] = $I0
        LL-END3:
        LL-END1:

SetField( Px, f, Py, false) =
            $P0 = Px["f"]
        IsType( $P0, atom, false)
            Px["f"] = Py

SetFieldValue( Px, f, Py, Sy, false) =
            Px = Px["f"]
            $S0 = Px["_t"]
            if $S0 != Sy goto false
            if Sy != "str" goto LL-ELSE
            $S0 = Py["_v"]
            Px["_v"] = $S0
            goto LL-END
        LL-ELSE:
            $I0 = Py["_v"]
            Px["_v"] = $I0
        LL-END:
```

```
Value ( Px , focus , false ) =
                              Focus ( Px , focus , false )
Value ( Px , fieldselection , false ) =
                              SelectField ( Px , fieldselection , false )

CompareValue ( Px , int , v , false ) =
                                   $I0 = Px [ "_v" ]
                                   if $I0 != v goto false
CompareValue ( Px , bool , true , false ) =
                                   $I0 = Px [ "_v" ]
                                   unless $I0 goto false
CompareValue ( Px , bool , false , false ) =
                                   $I0 = Px [ "_v" ]
                                   if $I0 goto false
CompareValue ( Px , str , v , false ) =
                                   $S0 = Px [ "_v" ]
                                   if $S0 != PrintValue ( v ) goto false

ComparePValue ( Px , Py , false ) =
                                   $S0 = Px [ "_t" ]
                                   $S1 = Py [ "_t" ]
                                   if $S0 != $S1 goto false
                                   if $S0 != "atom" goto LL - ELSE1
                                   ne_addr Px , Py , false
                                   goto LL - END
                              LL - ELSE1 :
                                   if $S0 != "str" goto LL - ELSE2
                                   $S0 = Px [ "_v" ]
                                   $S1 = Py [ "_v" ]
                                   if $S0 != $S1 goto false
                                   goto LL - END
                              LL - ELSE2 :
                                   $I0 = Px [ "_v" ]
                                   $I1 = Py [ "_v" ]
                                   if $I0 != $I1 goto false
                              LL - END :
```

And finally, we describe some functions that operate on values.

PrintValue(v)
: Turns the string v into something printable (escaping \ and ", change a newline into \n, and puts it inside ").

Type(v)
: Determines the type of the value v.

DefaultValue(t)
: Returns the default value for type t (0 for int, 0 (false) for bool, "" for str).

4.4 Implementation

We implemented our compiler-projection as part of the PGA Toolset [9]. As is done with other tools in the Toolset, the compiler-projection is implemented in a generic form such that modules for projecting primitive and basic instruction sets can be loaded on request by the user. Figure 1 shows the import relations (that are of interest) of the different modules for our compiler-projection. The step `pglec2pgleca` is done internally by the parsing and checking of the PGLEc code, and together with `pglec-mspio2pir` correspond to `gencompiler`. The ψ operations correspond to `PGLEc2PIR` and the ϕ operations to the modules `MSPio2PIR` and `HMPPV2PIR`. The auxiliary functions used in the ψ and ϕ operations correspond to the module `Fluid2PIR`. The larger part of these functions is implemented as

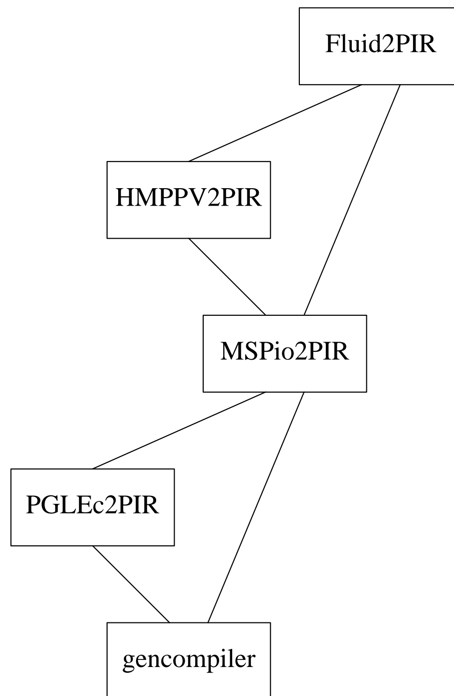

Figure 1. Model of the generic compiler-projection

macro's for PIR. This not only makes the generated code much more compact, but also made the development of our compiler-projection much easier.

4.5 Testing

Most of the testing is done with the use of small examples and inspection of the resulting code. But as a larger test that makes it possible to use the PGA Toolset in testing, we developed a projection of PGLB to PGLA in PGLEc.MSPio. The listing of this program, b2a.ec, can be found in Appendix A. This program can be simulated with the following command.

```
% gensim -P PGLEc -B MSPio -v -l b2a.ec
```

Put a PGLB program in focus INPUT and run the program. The result will be a PGLA program in focus OUTPUT.

We can compile our program with the command:

```
% gencompiler < b2a.ec > b2a.pir
```

Now we project our program in PGLEc to a program in PGLB.

```
% project pglec pglb < b2a.ec > b2a.b
```

We execute the compiled program with the result from the previous command, which gives us a version of our projection in PGLA.

```
% parrot b2a.pir < b2a.b > b2a.a
```

Now we can simulate our projection of PGLB to PGLA again, but this time with PGLA as primitive instruction set.

```
% gensim -P PGLA -B MSPio -v -l b2a.a
```

Another way to test our projection is by bisimulation. We first translate the versions of our projection in PGLEc and PGLA into labeled transition systems (lts).

```
% prog2lts -P PGLEc b2a.ec > b2a-ec.lts
% prog2lts -P PGLA b2a.a > b2a-a.lts
```

After which we can test for bisimulation.

```
% bisim b2a-ec.lts b2a-a.lts
```

5. Evaluation

We fulfilled our goal to project a language based on Program Algebra to an executable form. We did not strive to develop the best code generator possible, because the emphasis lay on the code generation process and not on the quality of the generated code.
The code generation process was rather easy. This is partly due to the simplicity of our programming language, but also to the execution model we used, which made it possible to design the process in a top down manner.

As our test showed, the I/O extension of MSP is satisfying. The instructions are powerful, but are still low level enough to make it possible to translate them directly to PIR.

Parrot turned out to be very suitable as target for our simple programming language. Because we already modeled the fluid in Perl for the simulator of the PGA Toolset and Parrot supports the data-types of Perl, implementing a model of the fluid in Parrot was easy. We must admit that Perl hashes are ideal for modeling the fluid, so it may take much more effort to implement a model for a basic instruction set that is not based on molecular dynamics.

Whether Parrot is the virtual machine to be used as prime target in future, we cannot say at this moment. For that, we should gain more experience with other languages that require more features of the virtual machine. And of course, other virtual machines such as .NET, JVM (Java Virtual Machine), should be considered also.

References

[1] *Parrot website*. http://www.parrotcode.org/

[2] *Perl website*. http://www.perl.org/

[3] *Program Algebra website*. http://www.science.uva.nl/research/prog/projects/pga/

[4] *Python website*. http://www.python.org/

[5] *Ruby website*. http://www.ruby-lang.org/

[6] J.A. Bergstra and I. Bethke, "Molecular dynamics," *Journal of Logic and Algebraic Programming*, vol. 51, no. 2, pp. 193-214, 2002.

[7] J.A. Bergstra and M.E. Loots, "Program algebra for sequential code," *Journal of Logic and Algebraic Programming*, vol. 51, no. 2, pp. 125-156, 2002.

[8] B. Diertens, "Molecular Scripting Primitives," report PRG0401, Programming Research Group - University of Amsterdam, 2004.

[9] B. Diertens, "A Toolset for PGA," report PRG0302, Programming Research Group - University of Amsterdam, 2003.

[10] A. Randal, D. Sugalski, and L. Tötsch, *Perl6 and Parrot Essentials,* second edition, O'Reilly & Associates, Inc., 2004.

Appendix A. Projection of PGLB to PGLA in PGLEc.MSPio

In the following, a listing is given of the program in PGLEc with MSPio to convert a program in PGLB to a program in PGLA.

```
  1 instrlist = null;
  2 nrinstr = 0;
  3 L0;
  4 instr = "";
  5 - read h {;
  6    ##L100;
  7 };
  8 L10;
  9 + h == " " {;
 10    read h;
 11    ##L10;
 12 };
 13 + h == "\n" {;
 14    read h;
 15    ##L10;
 16 };
 17 space = "";
 18 ##L13;
 19 L11;
 20 + read h {;
 21 L13;
 22    + h == " " {;
 23       append space " ";
 24       ##L11;
 25    };
 26    + h == "\n" {;
 27       append space " ";
 28       ##L11;
 29    };
 30    + h == "\"" {;
 31       append instr space;
 32       append instr h;
 33       space = "";
 34       L12;
 35       read h;
 36       append instr h;
 37       + h == "\\" {;
 38            read h;
 39          append instr h;
 40          ##L12;
 41       };
 42       - h == "\"" {;
 43          ##L12;
 44       };
 45       ##L11;
 46    };
 47    - h == ";" {;
 48       append instr space;
 49       append instr h;
 50       space = "";
 51       ##L11;
 52    };
 53 };
 54 + instrlist == null {;
 55    pinstr = new;
 56    instrlist = pinstr;
 57 }{;
 58    pinstr.+next = new;
 59    pinstr = pinstr.next;
 60 };
 61 + instr == "!" {;
 62    pinstr.+end;
 63    ##L20;
 64 };
 65 first instr h;
 66 + h == "+" {;
 67    pinstr.+testt;
 68    delfirst instr;
 69    L21;
 70    first instr h;
 71    + h == " " {;
 72       delfirst instr;
 73       ##L21;
 74    };
 75    pinstr.+basic:str = instr;
 76    ##L20;
 77 };
 78 + h == "-" {;
 79    pinstr.+testf;
 80    delfirst instr;
 81    L22;
 82    first instr h;
 83    + h == " " {;
 84       delfirst instr;
 85       ##L22;
 86    };
 87    pinstr.+basic:str = instr;
 88    ##L20;
 89 };
 90 + h == "#" {;
 91    delfirst instr;
 92    + int instr s {;
 93       pinstr.+goto:int = s;
 94       ##L20;
 95    };
 96 };
 97 + h == "\\" {;
 98    delfirst instr;
 99    first instr h;
100    + h == "#" {;
101       delfirst instr;
102       + int instr s {;
103          pinstr.+gotob:int = s;
104          ##L20;
105       };
106    };
107 };
108 pinstr.+basic:str = instr;
109 L20;
110 incr nrinstr;
111 ##L0;
112 L100;
113 pinstr = instrlist;
114 ic = 0;
115 + pinstr == null;
116 ##L200;
117 L110;
118 incr ic;
119 + pinstr/goto {;
120    h1 = ic;
```

```
121     incr h1 pinstr.goto;
122     h2 = nrinstr;
123     - decr h2 h1 {;
124        pinstr.goto = 0;
125        ##L190;
126     };
127 };
128 + pinstr/gotob {;
129     h1 = pinstr.gotob;
130     - decr h1 ic {;
131        h1 = nrinstr;
132        incr h1 2;
133        decr h1 pinstr.gotob;
134     }{;
135        h1 = 0;
136     };
137     pinstr.-gotob;
138     pinstr.+goto:int = h1;
139     ##L190;
140 };
141 L190;
142 + pinstr/next {;
143     pinstr = pinstr.next;
144     ##L110;
145 };
146 L200;
147 pinstr.+next = new;
148 pinstr = pinstr.next;
149 pinstr.+goto:int = 0;
150 pinstr.+next = new;
151 pinstr = pinstr.next;
152 pinstr.+goto:int = 0;
153 incr nrinstr 2;
154 pinstr.+next = new;
155 pinstr = pinstr.next;
156 pinstr.+repeat:int = nrinstr;
157 pinstr = instrlist;
158 ic = 0;
159 L300;
150 + pinstr/end {;
151     instr = "!";
152     ##L310;
153 };
154 + pinstr/testt {;
155     instr = "+ ";
156     append instr pinstr.basic;
157     ##L310;
158 };
159 + pinstr/testf {;
160     instr = "- ";
161     append instr pinstr.basic;
162     ##L310;
163 };
164 + pinstr/goto {;
165     instr = "#";
166     str pinstr.goto s;
167     append instr s;
168     ##L310;
169 };
170 + pinstr/repeat {;
171     instr = "\\\\#";
172     str pinstr.repeat s;
173     append instr s;
174     ##L310;
175 };
176 instr = pinstr.basic;
177 L310;
178 - ic == 0 {;
179     write "; ";
180 };
181 write instr;
182 incr ic;
183 + pinstr/next {;
184     pinstr = pinstr.next;
185     ##L300;
186 };
187 write "\n";
```

The lines 1 through 112 parse the input and build a list of instructions in focus `instrlist`. The list of instructions in PGLB is converted to a list of instructions in PGLA in the lines 111 through 156. The remainder of the code writes the list of instructions on the output.

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0403] B. Diertens, *A Compiler-projection from PGLEc.MSPio to Parrot,* Programming Research Group - University of Amsterdam, 2004.

[PRG0402] J.A. Bergstra and I. Bethke, *Linear Projective Program Syntax,* Programming Research Group - University of Amsterdam, 2004.

[PRG0401] B. Diertens, *Molecular Scripting Primitives,* Programming Research Group - University of Amsterdam, 2004.

[PRG0302] B. Diertens, *A Toolset for PGA,* Programming Research Group - University of Amsterdam, 2003.

[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/