Newgen User Manual

Pierre Jouvelot
Rémi Triolet

CRI, Maths & Systems,
MINES ParisTech,
35, rue Saint-Honoré,
77305 Fontainebleau CEDEX

December 1990
last modified: March 2, 2016 *(r1357)*

1 Presentation

The design and writing of large software generally requires the definition and implementation of application specific data structures. Newgen is a programming tool that helps in the second part of this process. From the high-level specification of user data types, it generates creation, access and modification functions that implement these data types, seen as abstract data types. In particular, functions to read and write Newgen-defined values on files are supported, thus providing interlanguage compatibility via files or streams. For instance, data structures created by a C program that uses Newgen-C and written on a file can be read by a CommonLISP program that uses Newgen-Lisp.

Newgen allows the definition of *basis domains*. These basis domains are the core on which more elaborate data types can be defines. For instance, Newgen provides `int` and `float` basis domains. On these predefined domains, Newgen allows the definition *constructed domains*; *product domains* allow the manipulation of tuples, *sum domains* tagged values (reminiscent of Pascal variant types), *list domains* list of values and multidimensional *array domains* fast direct-access set of values. Moreover, to allow an upward-compatible use of Newgen, *external domains* that permit the introduction of values from preexisting domains inside Newgen-generated data structures are supported.

From such definitions, Newgen generates, for each supported language[1], a set of functions that :

- create either initialized or not data values,
- access relevant parts of constructed values (e.g., elements in a list or projection on tuples),
- modify certain parts of constructed values,
- write and read values to and from files,
- recursively deallocate data values (if required by the underlying language).

This manul describes the specification language used by Newgen (section 2), introduces a simple example which will be used throughout this paper in examples (section 3), presents the C (section 4) and CommonLISP (section 5) version of Newgen before concluding with the practical aspects of Newgen (section 6).

2 The Newgen Grammar

The grammar of Newgen specifications, given in a Yacc-like syntax, is the following :

```
Specification :  Imports Externs Definitions

Imports : Imports IMPORT NAME FROM " NAME " ;
        |

Externs : Externs EXTERNAL NAME ;

Definitions : Definitions Definition
            | Definition

Definition : Tabulated NAME = Domain ;

Tabulated : TABULATED
          |
```

[1] C and CommonLISP are the currently supported languages

```
Domain  : Simple Constructed
        | { Namelist }

Simple  : Persistant Basis
        | Persistant Basis *
        | Persistant Basis {}
        | Persistant Basis Dimensions

Persistant : PERSISTANT
           |

Basis   : NAME
        | NAME : NAME

Constructed : x Simple Constructed
            | + Simple Constructed
            | Simple -> Simple

Namelist : NAME , Namelist
         | NAME

Dimensions : [ INT ] Dimensions
           | [ INT ]
```

where terminals should be obvious. Commented lines are signalled by a beginning `-`. Note that CPP (the Unix C preprocessor) commands can be used everywhere in this file (e.g., to give array dimensions in symbolic form); they will be automatically inserted into Newgen-generated files.

An *imported* domain is any domain defined (in another file) with Newgen. Newgen allows multiple-files specifications; any Newgen domain referenced but not defined inside a specification file as to be declared as an imported domain. The quoted name is the file in which the imported domain is defined. Currently, nested imports are not supported.

An *external* domain is used to introduce any data structure defined outside of Newgen. This facility allows the incremental use of Newgen inside projects that already have defined data structures or the integration of more sophisticated data structures (e.g., hash-coded data).

Newgen domains are either tabulated or not. A *tabulated* domain has its elements that are automatically associated to a unique identifier. This

identifier will be used to write in and read from files tabulated values, those providing for ways of naming values accross different program runs.

A basis domain is a labeled name `a:b` (as a short cut, `a` is equivalent to `a:a`) where `b` is a built-in domain known by Newgen. Supported basis domain are `unit, bool, char, int, float` and `string`. The type `unit` has only one value. When used inside more complex domains, a basis value is always *inlined* so that no space is lost by using Newgen with predefined types. For instance, lists of integers are not list of pointers to integers but are somewhat isomorphic to the kind of structure definition a C programmer would have thought of.

A *simple* domain is either a basis one, a list `*` of basis, a set {} of basis or a multidimensional array `[]` of basis. When a simple domain is tagged as *persistant*, it won't be freed when recursively reached by a deallocation expression.

Constructed domains can either be a product `x` or a sum `+` of simple domains; the notation {`a,b`} is equivalent to `a:unit+b:unit`. A sum domain is the equivalent of Pascal variant types and is used to represent in a single type values from different types. A product domain is the equivalent of Pascal record types and is used to represent in a single type a collection of values. The arrow type `->` is only available in C mode and allows functions to be defined.

Domain definitions are mutually recursive inside one specification file.

For instance, here are three different definitions of `list_of_names` where each name is implemented by the (predefined) `string` data type: the first one use the Newgen list constructor, the second explicits the recursive definition of list and the third uses an array to store names:

```
- Built-in list
-
list_of_names_1 = names:string *

- Explicit definition
-
list_of_names_2 = nil:unit + not_empty_list_of_names
not_empty_list_of_names = name:string x reste:list_of_names_2

- Array implementation
-
#define MAX_NAMES 100
list_of_names_3 = names:string[ MAX_NAMES ]
```

Values created with Newgen primitives are first-class. They can be passed to functions (the passing mechanism is a call-by-sharing reminiscent of Lisp), stored in data structures and assigned to data variables.

3 An example

We will use the following example specification throughout the whole paper. It defines a set of domains to be used in the implementation of a car reservation system used in our lab. For pedagogical purposes, this specification will be split across two files. The first file is `reservation.newgen`:

```
- File: reservation.newgen
-
- The CRI car reservation system specification
-
- French cars are small !
#define NB_OF_PASSENGERS 3

import location from "location.newgen"

external login

indisponibility = reservation*

reservation = driver:person x date x destination:location x
              nbpassager:int x
              passager:person[NB_OF_PASSAGERS] x
              to_confirm:bool

person = name:string x login

tabulated date = day:int x month:int x year:int x period
period = { morning , afternoon , day }
```

while the second is `location.newgen`:

```
- File: location.newgen
-
- Since the Ecole des Mines is decentralized, locations
- can vary !
-
```

```
location = known + other:string
known = { paris, sophia, rocquencourt }
```

.

In this example, dates will be tabulated. A `login` is an external data type that is supposedly manipulated by previoulsy defined functions. An `indisponibility` defines the already occupied slots for the (unique, our lab is not that rich) car. Each of these `reservations` defines a bunch of useful information among which the `date` and a boolean specifying whether a confirmation is requested or not. The other domains should by now be obvious.

4 Newgen-C

This section describes the C implementation of Newgen. Except otherwise specified, tabulated and untabulated domains are treated in the same way.

4.1 Creation Functions

Basis domains have been `typedef`-ed. Integer, float, character and string (i.e. `char *`) literals can be used, as well as `TRUE` and `FALSE` for booleans and `UU` for `unit` values.

From the definition of a domain `a`, the function to be used to create a value of that type is

```
make_a( <list-of-arguments> )
```

There are as many arguments as there are subdomains in the definition of the domain `a`. *Example*: `person p = make_person("Foo", login_foo );` creates a person `p` with name `"Foo"` and login `login_foo` supposedly defined outside. Undefined values can also be used; for each domain `a`, an undefined value `a_undefined` is provided. Also the functions `a_undefined_p`, which tests whether its argument is undefined, and `copy_a`, which recursively copies its argument (up to tabulated domains and persistant simple domains), are provided.

The list of arguments is empty for an arrow type `f = a -> b` . A new arrow object can be created by extendign a previous one via the `extend_f` function.

Lists are made of *cons* cells. To create a cons cell, use the macro `CONS` that returns a value of type `cons *`. You can use `NIL` to denote the end of a list. You have to provide the type of the element you cons. *Example*: To

create a one-element list containing person `p` use `cons *l = CONS( PERSON, p, NIL );`.

An external value cannot be created within Newgen.

4.2 Access functions

For a domain `a` that is constructed with a subdomain `b`, use the macro `a_b` to access the `b` field from an `a` value. *Example*: `string s = person_name( p );` returns the name of a person `p` in the string `s`.

For sum domains, the function `or_tag` accesses the tag (of type `tag` which is typedefed). Tags from `a = b+c` are `is_a_b` and `is_a_c`. Integer operations can be used on tags. To check the value of a tag, the boolean function `a_b_p` can be used. *Example*: To dispatch on the tag of the period `r`, use `tag rt = or_tag( r ) ; if( rt == is_period_morning ) ....` The C `switch` construct can also be used.

For the arrow domains `f = a->b`, i.e. for mappings, the function `apply_f` is defined. It takes a function and an object of type `a` and returns an object of type `b`.

To access list elements, use the `CAR` and `CDR` macros. You have to indicate the type `a` of the element with the macro `A` when you use `CAR`. *Example*: To get the second element of a list `l` of persons, use `PERSON( CAR( CDR( l )))`. A library of useful Lisp-like functions is provided; look for usage in the files `list.h` and list.c.

To acess set elements, use the set access functions that are exported by the `set` module; see `set.h` and `set.c` for a list of functions. Note that this specification of sets doesn't allow the C assignment but requires the function `set_assign` to be used. Thus C aliasing is prohibited. Set elements must be accessed with the `SET_MAP` macro. *Example*: To print the integers from a set `s`, use `SET_MAP( i, {printf( format, i );}, s )`.

To access array elements, use the usual C brackets. You have to indicate the type `a` of the element with the macro `A`. *Example*: To get the second element of an array `t` of passengers, use `person second = PERSON( t[1] );`

External values and variables can be declared in the same way. You have to cast your (addresses to) exogeneous values to the external type. *Example*: If a login is defined by `struct mylogin`, then use

```
struct mylogin {char *name ; int id;} joe ;
joe.name = "Joe Luser" ;
joe.id = 999 ;
person_login( second ) = (login)&joe ;
```

For debuygging purposes, you can check the validity of a NewGen data structures by calling the function `gen_consistent_p`. Also, the function `gen_sharing_p` returns 1 if its two arguments have a pointer (either a Newgen value or a CONS cell) in common.

When recursively traversing a Newgen value, one can use the general parametrized traversal routine `gen_recurse`, which is a simplified version of `gen-recurse` used in the CommonLISP mode. It allows for easy definition of top-down and bottom-up algorithms on Newgen value:

```
void
gen_recurse( obj, domain, filter, rewrite )
"Newgen object" obj ;
int domain ;
bool (*filter)( <objet Newgen> ) ;
void (*rewrite)( <object Newgen> ) ;
```

The `object` is recursively traversed (until basis domains, except through persistant arcs or tabulated domains). For any object `x` in the `domain` present in `obj`, the `filter` function is applied. If false is returned, the traversal of `x` is not performed. Otherwise, once the recursive traversal of subobjects of `x` is performed, the `rewrite` function is applied, taking `X` as an argument.

For any domain `d` defined in Newgen, the macro `d_domain` is defined.

An extended version of this function is also available to traverse a Newgen value with top-down filtering decisions and bottom-up rewriting actions on several domains. The traversal is optimized so that only useful arcs are followed, *i.e.* arcs that may lead to a domain to be visited.

```
void
gen_multi_recurse(<Newgen Obj> obj,
 [int d, bool (*flt)(<Newgen Obj>), void (*rwt)(<Newgen Obj>)]*,
            NULL);
```

4.3 Modification functions

To modify a value, use the access function in the left hand side of an assignment. *Example*: To change the name of the second person, use `person_name( second ) = "new_name";`.

For arrow types, i.e. for mappings, use `update_f` or `extend_f` for the domain `f`.

For sets, use `set_assign` or the triadic operations defined in `set.c`.

4.4 Destruction function

To free a (non external) value created by a `make` function, use `gen_free`; no value is returned. Values are recursively freed (except when an undefined member, an inlined value or a tabulated value is encountered). A tabulated value has to be explicitly freed by a direct call to `gen_free`. *Example*: To free a person `p`, use `gen_free( p )`.

4.5 IO functions

You can output (non external) Newgen values on a file with `gen_write` and read them back with `gen_read`. In the process, sharing and circular lists are preserved. *Example*: To output a person `p` on a FILE * `fd`, use `gen_write( fd, p )`. To read it back, use `p = (person) gen_read( fd );`.

These IO functions recusively write and read their subcomponents in the obvious way, except for tabulated domains. A value from a tabulated domain, when indirectly reached, is not actually written on the file; the only information actually output is the unique identifier of the value (plus some Newgen information). To effectively write a tabulated value, you have to call `gen_write` directly on this value.

When read back, an identifier from a tabulated domain will refer to the actual value; this value will have to have been previously read by a call to `gen_read`.

4.6 How to use Newgen-C

Once Newgen is installed, call `newgen`with the `-c` option followed by the names, here `reservation.newgen`, of the file(s) that contain(s) your specification; the specification files must form a complete specification. For each file (e.g. `reservation.newgen`), this creates one file: `reservation.h`. `reservation.h` contains all the definitions of the functions described above (this file has to be included in your C program). You have to call `gen_read_spec` in your C program before using any of Newgen generated functions; the order of the arguments (such as `reservation_spec`) is output by `newgen`.

For each external type, you have to call `gen_init_external` with five arguments: the type (in upper case) with `_NEWGEN_EXTERNAL` appended), a read function that receives as argument the function to read characters from, a write function that expects a `(FILE *)`, a free function that receives an object of the external type and a copy function that receives an object of the external type and returns a freshly allocated copy of it.

A program that uses Newgen functions has to include the file `genC.h`.

Example: To use our previous example, write:

```
#include <stdio.h>
#include "genC.h"

/* location is first is it is used by reservation */

#include "location.h"
#include "reservation.h"

/* We use ANSI-C function prototypes */

login read_login( int (*read)()) ;
void write_login( FILE *fd, login obj ) ;
void free_login( login obj ) ;
login copy_login( login obj ) ;


main()  {
        gen_read_spec( location_spec, reservation_spec, NULL ) ;
        gen_init_external( LOGIN_NEWGEN_EXTERNAL,
                           read_login, write_login,
                           free_login, copy_login ) ;
        {
        person p = make_person( "Foo",
                                (login)read_login( getchar )) ;

        /* Write user code here */
        }
}
```

To run this program, you have to link it with the `genC.a` library.

If you want to perform run-time checking of the consistency of your data structures, compile your C program with the `-DGEN_CHECK` option (this may generate incorrect code - especially in nested function calls - so you should always run "lint" on your program and add supplementary parentheses where necessary) and position the `gen_debug` variable to `GEN_DBG_CHECK`. To get more information about the way Newgen functions manipulate your data, use `GEN_DBG_TRAV` in the `gen_debug` variable. Various debug flags can be ored in `gen_debug` (see `genC.h`),

Run-time checked modules can be used with non-checked modules.

5 Newgen-Lisp

This version describes the CommonLISP implementation of Newgen. All the defined values in a file are external (in Lisp terminology) to the package `file`; use `use-package` to easily access them. You may have to manage package conflicts with the `shadow` function.

5.1 Creation Functions

From the definition of a domain `a`, the function to be used to create a value of that type is

```
(make-a <list-of-arguments>)
```

with the syntax of CommonLISP defstructs. *Example*: `(setf p (make-person :name "Foo" :login login-foo ))` creates a person `p` with name `"Foo"` and login `login-foo` supposedly defined outside.

For every non-inlined type `a`, an undefined value can be obtained by calling `make-a` without arguments.

Lists are made of Lisp cons cells. *Example*: To create a one-element list containing person `p` use `(setf l (list p))`.

An external value cannot be created within Newgen.

5.2 Access functions

For a domain `a` that is constructed with a subdomain `b`, use the macro `a-b` to access the `b` field from an `a` value. *Example*: `(setf s (person-name p ))` returns the name of a person `p` in the string `s`.

For sum domains, the function `or-tag` access the tag. Tags from `a = b+c` are `is-a-b` and `is-a-c`. Integer operations can be used on tags. To check the value of a tag, the boolean function `a-b-p` can be used. You can also use the `switch` macro accessible from the Newgen-Lisp library. *Example*: To dispatch on the tag of the period `r`, use

```
(if (= (or-tag r)} is-period-morning) ...)
```

With the Newgen `switch` macro:

```
(switch (or-tag r)
        (is-period-morning ...)
        (:default ...))
```

If no default is provided, a break is executed in case no tag matches. Insted of a tag as head of a clause, a two-element list (e.g., `((is-a-b b) ...)`) can be used; in that case, inside the body of the clause, `b` will be bound to the result of calling `a-b` on the expression used in the `switch` macro.

To access list elements, use Lisp. *Example*: To get the second element of a list `l` of persons, use `(cadr l )`.

To access set elements, use the module `set.cl` that uses the same specification as `set.c` (see above).

To access array elements, use the usual Lisp brackets `aref`. *Example*: To get the second element of an array `t` of passengers, use `(setf second (aref t 1))`.

NewGen also provides the `gen-recurse` macro described in the paper "Recursive Pattern Matching on Concrete Data Structures" (with B. Dehbonei), SIGPLAN Notices, ACM, Nov.89. This macro allows recursive programs over recursively defined data structures to be written very simply. See the documentation string of `gen-recurse` for a rapid description.

External values and variables can be used in the obvious way.

5.3 Modification functions

To modify a value, use the `setf` Lisp modification function. *Example*: To change the name of the second person, use `(setf (person-name second) "new-name")`.

For sets, use `set-assign` or the triadic operations.

5.4 Destruction function

No need for that, the GC will do it for you, except for tabulated values (they have to be explictly freed by a call to `gen-free`).

5.5 IO functions

You can output (non external) Newgen values on a file with `gen-write` and read them back with `gen-read`. In the process, sharing and circular lists are preserved. *Example*: To output a person `p` on a stream `st`, use `(gen-write st p)`. To read it back, use `(setf p (gen-read st))`.

See the same subsection in Newgen-C for a discussion about tabulated domains.

5.6 How to use Newgen-Lisp

Once Newgen is installed, call `newgen` with the `-lisp` option followed by the name, here `reservation.newgen`, of the file(s) that contain(s) your specifications. For each file, this creates one file, e.g. `reservation.cl`. `reservation.cl` contains all the definitions of the functions described above (this file has to be loaded in your Lisp program).

For each external type, you have to call `gen-init-external` with three arguments: the type (in upper case), a read function that receives as argument the function to read characters from and a write function that expects a stream and a pointer to an object of the external type.

A program that uses Newgen functions has to load the file `genLisplib`.

Example: To use our previous example, write:

```
(load "genLisplib")
(load "location.cl")
(load "reservation.cl")

(use-package :reservation)

(defun main ()
        (gen-init-external login read-login write-login)
        (setf p (make-person :name "Foo"
                             :login (read-login read)))
        ; Write user code here
        )
```

You can also use compiled versions of these files.

6 Conclusion

Newgen is currently available on an "as is" basis in the Newgen-C and Newgen-Lisp versions from the authors. Written in C, it has been successfully used on Sun-3, Sun-4 and ATT-3B5. Requests and comments can be mailed to:

`jouvelot@cri.ensmp.fr`