A Generic Programming Environment for High-Performance Mathematical Libraries*

Wolfgang Schreiner, Werner Danielczyk-Landerl,
Mircea Marin, and Wolfgang Stöcher

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
FirstName.LastName@risc.uni-linz.ac.at

Abstract. We report on a programming environment for the development of generic mathematical libraries based on functors (parameterized modules) that have rigorously specified but very abstract interfaces. We focus on the combination of the functor-based programming style with software engineering principles in large development projects. The generated target code is highly efficient and can be easily embedded into foreign application environments.

Keywords: functors, specifications, computer algebra, generic libraries.

1 Overview

We describe the preliminary results of a project that pursues the development of a programming environment for the construction of generic mathematical libraries. The project employs *functors* (parameterized modules) as the basis for constructing multi-layered software libraries for solving problems in various mathematical domains. Our work in this direction has been inspired by previous attempts on building generic computer algebra libraries that were seriously hampered by a lack of appropriate language support [7, 12].

Module *parameterization* as the basic device for building large software systems has been pioneered by the functional language SML [1] and by special-purpose computer algebra languages [2, 8, 6]. Nevertheless, the main-stream in software engineering concentrates on the object-oriented notion of type *inheritance* for building reusable software [5]. However, it is difficult to model mathematically axiomatized structures by inheritance; parameterized structures offer a much more natural framework.

In industrial programming languages, parameterized program units have only found limited support. A major exception is the *template* feature in C++ which is the basis for the Standard Template Library [10]. However C++ templates do not provide means for specifying information about their parameters; thus the body of a template cannot be compiled (typed-checked) on its own. Another

* Supported by the Austrian Science Foundation FWF grant P11414-ÖTE "HPGP — High-Performance Generic Programming"

problem is the inability of many C++ implementations to automatically share instantiated code among different users of a template library. Both problems make templates difficult to use in large-scale software development [3]. On the contrary, SML's functors are statically type-checked; however, they are compiled to function tables which are looked up at runtime with corresponding performance overheads. Both C++ and SML do not provide means for the specification of a component's behavior (apart from its syntactic interface and type signature).

The goal of our project is to provide an environment for the multi-user development of generic mathematical libraries without compromising the performance of the generated code. Our approach is based on the following concepts:

Low-level core language We utilize a low-level core language for writing the executable entities (functions, procedures) that are embedded in and exported by modules. This core language does not provide any powerful builtin concepts that would require a complex runtime system; it is essentially a simple subset of C. Our goal is to build all abstractions necessary for convenient programming in the generic programming environment itself.

Powerful abstraction facilities The environment provides powerful facilities of constructing generic software libraries. *Specifications* define the syntactic and semantic interface of modules; *functors* describe how new modules can be constructed from other modules; *modules* are constructed by application of functors to modules; *packages* group specifications, functors, modules, and other packages together.

Efficient implementation The implementation is carefully designed to make sure that abstraction is not punished by runtime overhead. Information about the implementation of an entity is transparently propagated through all levels of module abstractions such that ultimately the same machine code is generated than without using the abstraction. To avoid code explosion, equivalent functor instantiations do not generate duplicate code.

Support for project management Packages may be jointly developed and utilized by multiple persons in a shared environment. The system is designed and implemented such that code is never unnecessarily duplicated, that permissions are set correctly, and that dependencies between different units are automatically maintained.

The last two items raise issues that are crucial for the practical success of a generic programming environment. If generic abstractions are penalized by performance loss, by code explosion, or by management overkill, programmers are tempted to avoid them. We have taken considerable efforts to overcome these problems in order to develop efficient code that is independent of any builtin types and type constructors, and that does not hardwire module dependencies.

In the following section, we will sketch our approach to tackle these requirements. Our goals are not yet completely fulfilled, because the environment is currently restricted to first-order functors, i.e., functors cannot be components of modules or passed as arguments to other functors. However, after the current status of the development has been consolidated, we intend to further develop the system into a higher-order framework [9].

2 The Programming Environment

We are going to explain the environment in an informal style demonstrating its most important features on a case-by-case basis. A systematic description and user manual will be soon available in [11].

2.1 Packages

Packages represent the basic device for imposing *structure* on libraries; they allow to group related program units (specifications, functors, modules, and packages themselves) under a common header. Packages also restrict references to units outside the package boundaries by import and export interfaces; this enables the package maintainer to control inter-package dependencies.

A package is created by compiling a description such as

```
package Arithmetic<>Algebra
{
  use package Basic;
  import all of Basic<>Standard;
  export spec Ring, Field;
  export module IntRing
}
```

which introduces a subpackage `Algebra` of package `Arithmetic`. By default, the only units that may be referenced by units in `Algebra` are those defined within the package itself. The `use` directive, however, also gives access to the units exported by `Basic` (which must be visible in parent `Arithmetic`). The `import` directive adds all units of package `Standard` to the environment of `Algebra` as if they were defined within this package. Units of other packages may only refer to those units of `Algebra` that are listed in the `export` directive.

Our package concept is strictly more flexible and powerful than C++ namespaces or Java packages: explicit interfaces restrict the use of units from other packages; units may be used, imported, and exported under a different name (decoupling the name of a unit from references to the unit); units imported into a package may be as well exported from that package (up to the point of completely "virtual" packages that do not contain actual code but collect units from other packages). Since package descriptions may cyclically refer to each other, mutual references of units from different packages are allowed.

Compiling a package description creates the physical directory structure required for installing units within this package. Its essential purpose however is to set up the name space that maps unit references to file paths and that is used by every unit of this package to get access to other units. Logical unit references are therefore strictly separated from physical file locations, which is important for managing the development and evolution of large libraries.

As set up by the package description, the name space is fixed for all units in this package; this also helps to establish a uniform convention for external references across the source code of a package.

2.2 Specifications

General A specification describes the syntactic and semantic interface of a module. A sketch of such a specification is shown below:

```
spec Standard<>Bool
{
  type Bool                              // types

  constr b: Bool()                       // constructors
  constr b: Bool(c: Bool)                // and destructors
  destr b: Bool()

  let type B builtin Bool                // mapping to
  in                                     // builtin boolean
  {
    fun trueValue(b: Bool): B
    axiom type Bool truth trueValue
  }

  fun operator ==(x: Bool, y: Bool): Bool  // equality on Bool
  axiom type Bool equality operator ==
  ...
  fun operator !(b1: Bool) b2: Bool      // negation
  output (b1 == true)  && (b2 == false)
      || (b1 == false) && (b2 == true)
  ...
  proc operator =(out b1: Bool, b2: Bool)  // assignment
  output b1 == b2
  ...
}
```

This specification `Bool` in package `Standard` introduces a type `Bool` with associated constructors/destructors, values, functions, and procedures (an explanation follows below). A specification may also contain modules as in

```
spec Array
{
  module I: Index
  module E: Element
  axiom type I::Bool = E::Bool
  import I
  import E except Type
  ...
  fun operator ==(a: Array, b: Array) =
    maxIndex(a) == maxIndex(b) &&
    forall(i: Index) (0 <= i && i < maxIndex(a) => a[i] == b[i])
}
```

An entity c within module M may be referenced as `M::c`; the current environment inherits the contents of a module by an `import` clause (optionally with restricting qualifiers such as `except` or `only`).

Blocks A block `let` S_1 `in` S_2 introduces in specification S_1 entities that may be used locally within specification S_2; however, only the contents of S_2 become part of the enclosing environment. Thus an implementation of this specification needs not provide an implementation of S_1; this construct therefore allows to express existential quantification in specifications.

Types A type is identified by a name; by default, different names denote different types. Functions/procedures with the same name but with different argument number or argument types can coexist in the same scope ("overloading").

The construct `axiom type` T_1 = T_2 constrains two names T_1 and T_2 to denote equal types; without this constraint, specification `Array` would presumably not type-check because `I::Bool` and `E::Bool` would denote different types. The construct `axiom module` M_1 = M_2 makes all types in M_1 equal to the corresponding types of M_2 (which must satisfy the specification of M_1).

Constructors and Destructors A constructor is a procedure associated to a type. The procedure is invoked by a declaration of a value/variable of this type. E.g,

```
var x: Bool(y)
```

calls the constructor `Bool` with argument `y` to initialize the `Bool` variable `x`. On exit from the scope, the destructor `Bool()` is invoked. Constructors may be also given explicit names which allows declarations such as `var x: Int‘one()`.

Special Axioms The construct `axiom type` T `truth` t specifies T as a truth type by stating that the function t maps any T object to a corresponding machine ("builtin") boolean. This specification enables us to write for a T object x a core language statement

```
if (x) ... else ...
```

which the compiler translates to

```
if (t(x)) ... else ...
```

In a similar way, we detach all core language constructs from machine types.

The construct `axiom type` T `equality` e declares e as the equality function on T. This states that, if e yields true on two arguments of type T, these arguments cannot be distinguished by any user of the module being specified (even if the internal representations of these objects are different).

Parameter Modes In procedure specifications, the parameter modes `in` (default), `inout`, and `out` specify whether the object denoted by the parameter is modified by the procedure, i.e., whether the object is used by the procedure and whether its value after the call equals its value before the call. A parameter mode does not yet imply a particular argument passing strategy; such a strategy is selected by the compiler at instantiation time from the mode and from the concrete parameter type. Thus in contrast to C++, if an `in` argument happens to be passed by value, just the argument's representation is copied but no constructor is invoked (the programmer is not able to modify the parameter).

Pre- and Post-Conditions Specifications may attach pre-conditions and post-conditions to functions and procedures. Such conditions embed expressions (of some truth type) that may be directly executable as in

```
fun operator /(x: Int, y: Int) z: Int
input y != 0 && y | x
output x == y*z
```

but may also embed universal or existential quantifiers as in

```
fun operator |(x: Int, y: Int) r: Bool
output r == (exists(z: Int) y == x*z)
```

In debugging mode, the executable (parts of) axioms are compiled to *assertions* that are implicitly invoked with every application of the corresponding entity.

Constructive Specifications From function specifications like

```
fun operator <=(i: Int, j: Int): Bool =
  (i < j) || (i == j)
```

the compiler generates default implementations provided that the right-hand side represents an executable expression. In a corresponding functor definition, the programmer may request to use this default implementation, or she may explicitly give an (equivalent but presumably more efficient) implementation.

Parameterized Specifications To increase its applicability, a specification may be also parameterized, as in

```
spec Arithmetic<>EuclidRing[Bool: Bool]
{
  include Ring[Bool]
  ...
}
```

which includes the content of another parameterized specification `Ring` (and is a syntactic shortcut for declaring a local module with this specification and importing its content).

Specification Checking As sketched above, specifications define the interface of a module. The compiler can check whether a module confirms to the static signature of a specification (consisting of the names and the types of the exported entities); it is not able to verify whether the module actually satisfies the dynamic axioms. However, the main purpose of axioms is to rigorously specify the most important aspects of a behavior of a module such that, e.g., an implementation can be *falsified* by a counter example. In the form of *assertions and default implementations*, axioms also have some computational significance.

2.3 Functors

General A functor constructs a result module from modules passed as arguments; both argument and result modules are constrained by specifications. E.g.,

```
functor Pair[X: Type, Y: Type]: Pair[X, Y]
{
  type X = X::Type
  type Y = Y::Type
  type R = Record[x: X, y: Y]
  type Pair new R
  ...
  constr p: Pair(x: X, y: Y) inline
  {
    constr repr p: R()
    constr (repr p).x: X(x)
    constr (repr p).y: Y(y)
  }
  reffun operator .x(inref p: Pair): X inline
  {
    return (repr p).x
  }
}
```

satisfies the specification

```
spec Pair[X: Type, Y: Type]
{
  axiom type X::Bool = Y::Bool
  type Pair
  constr p: Pair(x: X::Type, y: Y::Type)
  reffun operator.x(inref p: Pair): X::Type
  ...
}
```

for some modules X and Y that satisfy `Type` and have equal `Bool` types.

New Types Above functor constructs a "new" type `Pair` based on the representation type `Record[...]`. There are no operations, constructors, or destructors defined on the new type; however, the expression

```
repr r
```

maps an object r of this type to an object of the underlying representation type. Thus a `Pair` constructor can be defined that first calls the `Record` constructor and then initializes the components by explicit constructor calls of the form

```
constr exp: Type(args)
```

Then we can create a value or variable of some instance of `Pair` as

```
type P = Pair[Int, Char]::Pair
var p: P(172, 'a')
```

where `Pair[Int, Char]` denotes the application of functor `Pair` to modules `Int` and `Char`, which creates a module that exports a corresponding type. The constructor associated to this type is used to declare the variable p.

This example demonstrates how type constructors like "Pair", "Array", or "Pointer" can be implemented by functors; consequently the core language does not provide any such concepts as "builtin". The only restriction is that our environment does not yet allow functors with an arbitrary number of arguments, thus there is still the necessity of the builtin constructor `Record` shown above.

Reference Functions The reference function `operator .x` may be used as

```
val x0: Int = p.x
p.x = x0
```

to get access to this component. The compiler parses this construction as

```
val x0: Int(operator .x(p))
operator =(operator .x(p), x0)
```

exhibiting that the result of the reference function is used as an output argument for procedure `operator =`. If `operator .x` is not visible in the environment (by import of `Pair[Int, Char]`), the qualification `p.[P]x` denotes the module P where `operator .x` is embedded. The `inline` tag in the function definition (not specification!) ensures that the code is inserted for every application.

Generic Literals A literal such as "172" is essentially parsed as

```
operator 0d(#1, operator 0d(#7, operator 0d(#5)))
```

where `operator 0d` is a function and `#1`, `#7`, `#5` are values visible in the current scope. They are typically introduced by the import of a module that exports

```
val #0: Int
...
fun operator 0d(x: Int, y: Int)
```

which allows to use integer "literals" for type `Int`. Likewise, floating point literals (`1.72`) and string literals (`"abc"`) are not restricted to machine types (if defined by builtin operations, literal values are still computed at compile time).

Initializers and Terminators In order to setup the state of the result module (represented by globally declared variables and values), a functor may contain one ore more initializer and terminator statements (similar to Java). Initializers are automatically executed before the module is executed, e.g.,

```
val #0: Int
init
{
  constr repr #0: I`zero()
}
```

initializes the value `#0` by a call of constructor `zero` in representation type `I`. Different from C++, module initialization proceeds in dependence order, which ensures that the arguments of a functor instantiation are initialized before the result is initialized. Correspondingly, terminators ensure proper deinitialization.

Builtin Objects Since any module hierarchy has to be ultimately based on machine objects, we need an interface to refer to these. A definition

```
type I builtin Int
fun operator +(x: I, y: I): I builtin operator +
```

defines a function `operator +` by the builtin operation `operator +` with the denoted type signature. This interface is only intended for the implementation of a fixed set of low level modules whose internals are subsequently hidden.

External Interfaces In order to get access to code written in other programming languages, definitions like

```
fun operator +(x: I, y: I): I extern intSum
```

may be used. This generates an external reference to a function `intSum` for which an external object has to be provided and linked to the application program.

2.4 Modules

A module is composed by a nested functor application with references to other modules (possibly defined locally within the definition), e.g.

```
module Application<>Main: Main =
let
{
  module I: Int = Language::Int
  module B: Bool = Language::Bool
  module R: Rec = Pair[I, B]
} in
  Main[Language, R]
```

References to concrete modules (e.g., `Language`) are restricted to module definitions; functors can access modules only by their parameters. In the ultimate higher-order framework, we intend to restrict functor access in the same way such that a functor does only depend on its parameter interface (without "hardwired" functor dependencies).

We also allow *cyclic* module definitions such as in

```
module Cycle: Spec =
let
{
  module A: S1 forward
  module B: S2 = F1[A, G<>M]
  module A: S1 = F2[B, G<>N]
} in
  F3[A, B, G<>O]
```

where modules `A` and `B` depend mutually recursively on each other, i.e., each module can refer to the entities defined in the other module. Such cyclic dependencies are hardly good practice and should be avoided. However, in a couple of situations it is extremely hard to cope without them; we support this possibility for pragmatic reasons.

3 Implementation

File System Structure An environment variable holds the names of the directories that represent the root of the package hierarchy. When a program unit located in package P is compiled, a subdirectory is generated in the corresponding location:

```
/P                  package P
  /spec
    /S                 specification P<>S
  /functor
    /F                 functor P<>F
      /instances
        /0               instance 0 of functor P<>F
  /module
    /M                 module P<>M
  /package
    /P0                package P<>P0
```

The programming environment transparently maintains dependencies and automatically recompiles outdated units without any assistance from the programmer. This is an important issue in a generic context where many instances of a functor may exist and user-assisted management is practically impossible.

Symbol Tables Every program unit is compiled to a symbol table which holds all information known about every exported entity. For a specification, this is the name and the type of the entity plus all (executable) axioms attached to it. For a functor, this also includes the definition of each used entity, i.e.,

- by a functor parameter (no definition is available yet),
- as a builtin object (implemented by the compiler),
- as an external object (implemented as an external reference),
- by explicit code (translated to C++),
- by constructive axioms (converted to explicit code),
- by another entity (from which it receives its definition).

The last case is relevant for entities that are defined as an `alias` of (i.e., by just renaming) another entity.

Instantiation When a module definition is compiled, all denoted functor instantiations are resolved. If an up to date instance of the same functor with the same module arguments already exists, no code is generated for the module and the corresponding symbol table is read from file.

If a new instance of a functor has to be created, the symbols in the functor parameters (available from the functor symbol table) are temporarily linked to the corresponding symbols of the argument modules (available from the module symbol tables) thus resolving all definitions. A simplified duplicate of the functor symbol table is generated as the symbol table of the new module. Furthermore, the code generator produces C++ source files, from which the native compiler generates object code. For explicitly coded entities, the module symbol table does not contain the definition itself but just the information how this entity can be referenced (typically the external name).

Since instantiation transparently propagates all information that is available on the definition of a symbol, the efficiency of access to an entity is independent of the layers of abstractions that were put between the definition of the entity and its use: an entity defined as **builtin** remains known as builtin, every access to an entity defined with an **inline** tag is resolved by inserting this code. Unlike C++, the instantiation process is not limited by any object code boundaries!

Target Code A functor instance is compiled to a set of C++ (essentially C) include and source files which are in turn compiled by the GNU g++ compiler to object code. The include files contain type definitions, function prototypes, and function inline code for use by other instances. The interface to this instance is exhibited to the C++ level by compiling a module with option **-export**. This generates an include file for use in the application source code and a makefile that defines the list of all object files that have to be linked to the application.

The entities of an exported module are given externally visible C++ names as sketched in the following export header file of an instance of functor `Int`:

```
#ifndef Int_h
#define Int_h

/* module header file */
#include "/test/module/Int/instances/0/Int_0.h"

/* module initialization/termination */
#define init_Int() (_init_Int_0())
#define term_Int() (_term_Int_0())

/* types and their constructors/destructors: */
#define _type_Int_Int          _0_Int_0      /* type Int */
  #define _constr_Int_Int_0(o) ((void)0)     /* constr: Int() */
  #define _constr_Int_Int_1(o, x) ((o) = (x))/* constr: Int(Int) */
  #define _destr_Int_Int(x)    ((void)0)     /* destr: Int() */

/* procedures, functions, values and variables: */
#define Int_print ...                        /* proc(Int) */
#define Int__op_assign(x, y) ...             /* proc(out: Int, Int)*/

/* contents of exported modules: */
  /* types and their constructors/destructors: */
  #define _type_Int__Bool_Bool ...           /* type Bool */
  ...
#endif
```

The generated object code is of the same efficiency as if generated from a non-generic C++ program. It can be also conveniently used with symbolic debuggers, because pragmas refer to the correct locations in the functor source code.

4 Conclusions

Lack of space prohibits us from describing in more detail the specification and functor language, its type system, the implementation of instantiation, and of code generation [4]. The system has been developed in a very restricted subset of C++; it is reasonably stable such that first major experiments have become possible. As an important practical aspect, the environment still lacks an integrated revision control system; major missing language features are higher-order functors and functors with a variable number of arguments.

We have built a package `Standard` that specifies and implements the programming interface (i.e., the basic types and constructors) known from most imperative languages. This package consists of seven subpackages, about 80 specifications, 30 functors, and four module descriptions; all exported datatypes have been axiomatically specified yielding a number of default implementations and a large number of assertions (the assertion generator is net yet completed).

Based on the `Standard` package and on an `Extended` package for higher concepts such as reference-counted type constructors, we have started to build up a library for polynomial arithmetic that is intended to serve as a first major test bed and demonstration application for the system. We hope to show that the environment will be suitable for engineering highly reusable computer algebra libraries while preserving the utmost runtime efficiency.

References

1. D. Berry. Lessons from the Design of a Standard ML Library. *Journal of Functional Programming*, 3(4):527–552, October 1993.
2. CA Group. The Scratchpad II Computer Algebra Sytem Interactive Environment Users Guide. Draft 1.2, TJ Watson Research Center, Yorktown Heights, NY, 1988.
3. O. Caprotti, H. Hong, et al. On C++ Polymorphism for the STURM Library. Internal Note 94-001, RISC-Linz, 1994.
4. W. Danielczyk-Landerl. Syntax Processing and Code Generation for a Generic Programming Language. Master's thesis, RISC-Linz, 1998. To appear.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
6. H. Gast, S. Schupp, and R. Loos. Completing the Compilation of SuchThat v0.7. Technical Report 97-12, Rensselaer Polytechnic Institute, December 1997.
7. H. Hong, A. Neubacher, et al. The STURM Library Manual — A C++ Library for Symbolic Computation. Technical Report 94-30, RISC-Linz, April 1994.
8. R. D. Jenks. *Axiom — The Scientific Computation System*. Springer, Berlin, 1992.
9. X. Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95*, pages 142–153, San Francisco, CA, January 22–25, 1995. ACM Press.
10. D. R. Musser and A. Saini. *STL Tutorial & Reference Guide*. Addison-Wesley, Reading, MA, 1996.
11. W. Schreiner et al. HPGP User and Reference Manual. Technical report, RISC-Linz, 1998. http://www.risc.uni-linz.ac.at/projects/basic/hpgp, to appear.
12. W. Windsteiger and B. Buchberger. GRÖBNER: A Library for Computing Gröbner Bases based on SACLIB. Technical Report 93-72, RISC-Linz, 1993.