HPGP

High-Performance Generic Programming
for Computational Mathematics
by Compile-Time Instantiation
of Higher-Order Functors

Dr. Hoon Hong
Hoon.Hong@risc.uni-linz.ac.at
Tel. +43 7236 3231 48

Dipl.-Ing. Dr.[1] Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at
Tel. +43 7236 3231 66

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
Fax. +43 7236 3231 30

April 4, 1996

[1]Graduation *sub auspiciis praesidentis* is pending.

Abstract

A functor is a parameterized program module i.e. a function that takes modules as arguments and returns a module as its result. A higher-order functor deals in the same way with modules whose components are functors themselves. We propose to develop a generic compilation system for the construction of high-performance mathematical software libraries for scientific and technical application domains. This system has the following features:

1. It is based on a powerful higher-order functor language.
2. It is an open library that can be retargeted to any core language.
3. It is able to resolve functor instantiation at compile-time.

The functor language is expressive enough to build all types and type constructors without referring to the core language (thus maximizing flexibility) and to express all interactions between modules by parameterization (thus maximizing reusability). By compile-time instantiation, genericity does not cause any execution overhead; by automatically sharing instantiated code among all users, an explosion of library sizes is avoided.

Keywords: Genericity, modules, functors, mathematical libraries, software reuse.

Contents

List of Figures

Chapter 1

Overview

In this chapter, we give a summary of the proposal and sketch its main points.

1.1 Requirements

The development of scientific software libraries has during the recent years been hampered by the lack of programming systems in which source code may be written generically for a multitude of mathematical domains but that also generate efficient target code which may be applied in technical areas with high performance demands.

By the term "scientific software" we do not just refer to packages like LINPACK [Dongarra *et al.*, 1978] or LAPACK [Anderson *et al.*, 1995] that operate on very restricted domains (vectors and matrices of floating point numbers). On the contrary, this particular area has been well supported in the past yielding elegant and efficient libraries of numerical software.

Computational mathematics in general however deals with a much richer variety of structures such as arbitrary precision integers, rationals, complex numbers, modular numbers, algebraic numbers, univariate and multivariate polynomials, intervals, series, ...and then with sets, vectors, matrices, trees, ..., of such objects. While thus there exists an infinite number of mathematical types, mathematical programs can operate on whole classes of types that share some common properties.

Computer science has a large experience on how to develop programs and software libraries that operate on objects whose types are fixed; less experience exists on how to develop generic programs and software libraries that operate on objects of variable type. While most modern general-purpose programming languages support some notion of genericity, this notion is often too weak to express in a *safe* way the richness of mathematical structures. On the other hand specialized systems for computational mathematics have traditionally put less emphasis on performance than is required in real-world applications.

In 1993 Dr. Hoon Hong started at RISC-Linz a project to develop a software library STURM [Hong *et al.*, 1994] for solving systems of constraints. This work was based on his previous experience with the development of the C library SACLIB [Buchberger *et al.*, 1992]. Key issues of the STURM project were provably correct algorithms, genericity, and performance. An evaluation of available programming languages revealed that only C++ [Stroustrup, 1991, Stroustrup, 1994] provided the potential for developing efficient generic mathematical code; consequently this language was chosen for the implementation of the library.

While the STURM project has succeeded and the library will be soon released for public use, the developers had to undergo a lot of pain and suffering: the project revealed numerous semantic and pragmatic deficiencies of C++ that considerably hampered the project and lowered the productivity. In particular genericity turned out to be a major stumbling-block for the whole development process [Caprotti *et al.*, 1994]. As a whole, the project shed a disillusioning light on the available support for generic programming.

The proposed project is therefore is not only based on an abstract philosophy but also on real experience and concrete demands in an important area.

1.2 State of the Art

A variety of software packages has been developed using specialized systems for computational mathematics. In the recent past, especially the computer algebra system Mathematica [Wolfram, 1991] has been the basis of many mathematical libraries; other widespread packages are Maple [Char *et al.*, 1991] and Reduce [Rayna, 1987]. These systems are based on interpreters that execute proprietary dynamically typed programming languages. While genericity has originally not been a concern, generic programs may be nevertheless written by exploiting the higher-order nature of these languages [Monagan, 1993, Hearn and Schrüfer, 1993].

> At RISC-Linz the package CASA [Mnuk *et al.*, 1995] for constructive algebraic geometry was implemented in Maple; a Gröbner bases implementation has been developed in Mathematica [Buchberger, 1990]; in the Scratchpad II system [Scratchpad, 1988] a geometry theorem proving package was implemented [Kusche *et al.*, 1987].

This approach is adequate for reasoning, experimenting, and teaching but the performance of interpreted languages is not appropriate for developing mathematical libraries for high-performance applications.

Mathematical packages that are designed for intensive use are therefore typically written in industrially supported languages such as C [Kernighan and Ritchie, 1988]. The main advantage of this approach is the possibility to call libraries of such algorithms from application programs (by the standard linking interface) and the improved performance of the compiled language.

> At RISC-Linz the computer algebra library SACLIB [Buchberger *et al.*, 1992, Hong *et al.*, 1993] was developed to efficiently support arithmetic in various mathematical domains. The library is written in C and is non-generic i.e. it provides a separate implementation of every data type regardless of common mathematical properties.

Nowadays, C++ [Stroustrup, 1991] has taken the role of the mostly favored implementation language. There exist several C++ libraries e.g. on combinatorial and geometric computing (LEDA [Mehlhorn and Näher, 1995]), computer algebra [Kando, 1993], and constraint solving (STURM [Hong *et al.*, 1994]). Furthermore, there is a large supply of class libraries implemented by industrial vendors that provide general container classes such as lists, queues, hash tables, etc.

C++ supports two variants of expressing genericity. The original mechanism proposed for this purpose are *virtual classes* from which concrete instances are derived. However, this concept is implemented by lookup in function tables at runtime and thus too inefficient for expressing multiple levels of genericity.

> [Mehlhorn and Näher, 1995, Mehlhorn and Näher, 1992] report that generic programs based on C++ virtual functions and classes are penalized by a runtime increase of factor 2–10 (typically 4) and a memory increase of factor 2–3.5 compared to non-generic C versions.

Moreover, the inheritance concept on which object-oriented programming is based has several semantic problems that make it inappropriate for expressing complex generic constructions (see Section 3.3).

For supporting genericity without performance loss, C++ was later extended by the concept of *templates*. The concrete motivation for this extension was the need to express container types that are parameterized by a base type about nothing or only little needs to be known. From this background, C++ templates do not provide any means to specify information about template arguments. Consequently the application of templates can not be constrained and the correctness of a template definition cannot be checked on its own. These and other semantic and pragmatic deficiencies make templates hard to use for large-scale generic programming ([Caprotti *et al.*, 1994] and Section 3.2).

The core problem is that C++ has been developed for another purpose (low-level and higher-level systems programming, embedded systems, user interfaces and the like [Stroustrup, 1994]). While the language is very well suitable for these domains, it has not been designed for mathematical programming.

The genericity support of C++ and other industrially supported languages is inadequate for building high-performance mathematical software libraries.

1.3 The Functor Concept

A functor (in the sense of computer science) is a parameterized program module i.e. a function that takes modules as arguments and returns a module as its result. The terminology was introduced and the theory was elaborated in the programming language Standard ML (SML) [Milner, 1984, MacQueen, 1984, Milner *et al.*, 1990]. Generic program constructions are expressed within a statically typed framework i.e. every functor can be compiled and type checked on its own. SML modules are (similar to virtual classes in C++) implemented by function tables that are looked up at run-time; functors then correspond to runtime functions mapping function tables to function tables.

For this reason typical SML programs rely on functors only to a limited amount; most interactions between modules still rely on direct import and not on parameterization. However in contrast to C++ classes, the semantics of SML functors offers the *potential* for being implemented by code instantiation at compile-time (similar to the much less refined generic packages of Ada [Standard, 1983, Standard, 1994] and the templates of C++).

Another problem is that SML does not allow higher-order functors i.e. functors that deal with modules whose components are functors themselves. Without higher-order functors many important ways of modularization are not possible, in particular the *partial* instantiation of functors by which a functor library may provide various levels of generality for different demands [Berry, 1993, Appel, 1993].

However, recently the theory of higher-order functors has sufficiently matured [MacQueen, 1986, Harper *et al.*, 1990, Tofte, 1992, MacQueen and Tofte, 1994, Harper and Lillibridge, 1994] and a first experimental implementation of the concept was undertaken [Cregut and MacQueen, 1994].

While the *theory* of higher-order functors provides a very attractive foundation, there are several crucial points which prevent the actual use of the SML *system* for the purposes of mathematical programming:

1. SML functors are implemented as runtime functions, which allows only limited use of genericity if efficiency is important.
2. The SML module language is in all compilers tightly bound to the (mostly functional) SML core language and its implementation.

3. SML strictly separates types and type constructors of the core language from modules and functors of the module language.

The functor-based programming has (compared to the well-known object-oriented programming style) gained little popularity besides in parts of the computer algebra community [Watt *et al.*, 1988, Jenks and Sutor, 1992, Watt *et al.*, 1994]. However, this area is historically based on interpreted and dynamically typed languages where performance is not a primary goal. The situation is gradually changing by putting more emphasis on type systems [Santas, 1993, Zippel, 1993] and by the emerging of compilers for previously interpreted languages [Watt *et al.*, 1994]. Still these systems are not very efficient and based on proprietary core languages with comparatively little spread. We would like to overcome this situation by basing further developments on general purpose programming languages.

We propose to utilize the theory of higher-order functors for building generic software libraries in industrially supported programming languages.

1.4 Project Contents

We want to develop of a generic compilation system that incorporates the experience that RISC-Linz has gathered by the development of various mathematical libraries during the past years [Heinzelreiter *et al.*, 1987, Buchberger, 1991, Buchberger *et al.*, 1992, Windsteiger and Buchberger, 1993, Caprotti *et al.*, 1994, Mnuk *et al.*, 1995]. This system is intended to combine a clear and sound semantics (that facilitates the development of the software and the reasoning about it) with a high-performance implementation (that makes the results usable for real-world applications in science and engineering).

We do not propose to develop a new programming language.

Instead we want to construct an open library that can be easily retargeted to several industrially supported core languages e.g. C. *In this respect, our proposal is rather conservative and firmly based on the state of the art.*

We propose to develop a new generic compilation system.

The system will be based on higher-order functors that is much more powerful than corresponding concepts currently provided by industrially supported languages. In particular, the functor language will be rich enough to build all types and type-constructors (without referring to built-in types of the core language); it will be efficient enough to base all interaction between modules not on direct import but on parameterization. *In this respect, our proposal is very progressive and considerably exceeds the state of the art.*

Part of the proposed project will be the development of an interface to an existing core language (a subset of C) and the construction of a basic library of container types (lists, sets, etc.) and of mathematical objects (integers, rationals, polynomials). This is necessary to practically evaluate the adequacy of the system. We will also develop a precise specification and a formal semantics of the functor system. This will clarify the subtleties of its application and allow its integration with different core languages.

It will *not* be part of the project to build a completely new type theory (the essential theoretical concepts have been recently developed but still need adaptions and refinements), to build a new programming environment from scratch (we will integrate available tools as much as possible) or to use the system already to develop an advanced mathematical library (the porting of the STURM library will be the goal of a continuation project).

As demonstrated in this proposal we have already designed the overall structure of the system. We apply for financial support in order to proceed with the detailed design and to perform the actual implementation.

1.5 Researchers

The research institute RISC-Linz has an outstanding experience in symbolic and algebraic computation and its various application domains. The proposers of the project are well experienced in the development of mathematical libraries (required to understand the demands of the application area), in the formal semantics of programming languages (required to understand the subtle details of the functor-based system) and in the software-technological issues of compilation systems (necessary to successfully manage the implementation).

By close cooperation in a small but highly qualified working group, we will overcome the traditional diversification between research groups in computational mathematics and research groups in compiler construction.

We are thus highly confident to successfully fulfill the stated objectives.

1.6 Impacts

While the origin and the core interest of the proposed research is computational mathematics, we believe that the impacts of the project may far exceed this particular domain. The system will demonstrate that much of what is considered "hardwired" in programming languages (primitive types and type constructors) and in software libraries (the interaction between modules) can be effectively opened to the user. This will make software libraries much more useful by making them much more reusable.

Today the object-oriented programming style is very popular since it promises to increase reusability by building class hierarchies linked by inheritance relations. In fact, a whole software industry has emerged developing commercial class libraries for application domains like system programming, graphical user interfaces, client-server applications, etc.

In this proposal we promote a functor-oriented programming style that might well become equally popular for a large class of application domains, in particular but not restricted to those related to mathematics and computational sciences; it may thus even have more general impacts on the current understanding of software technology.

Computational mathematics serves as an extremely demanding "test-driver"; if the concepts prove useful in this domain, they may be applicable to other domains as well.

1.7 Proposal Structure

The rest of this proposal is structured as follows:

Chapter 2 (Requirements) describes our demands on a system for high-performance generic programming.

Chapter 3 (Concepts) sketches several concepts of genericity and justifies the choice of functors as the basis of this project.

Chapter 4 (State of the Art) discusses existing languages and systems with respect to our requirements.

Chapter 5 (Project Contents) presents the details of this project and how they meet the requirements.

Chapter 6 (Project Schedule) describes the organization and the schedule of the proposed project.

Chapter 7 (Researchers) sketches the qualification of the institute and of the researchers pursuing the project.

Chapter 2

Requirements

As sketched in Chapter 1, our ultimate goal is the development of libraries of mathematical software whose functions may operate on (infinitely) many different types of objects. These libraries are designed for the wide-spread use in application programs in various scientific and technical domains with generally high performance demands. For this purpose, we need a generic programming environment that fulfills the following overall objectives:

1. **Genericity.** It provides the necessary means to develop correct and elegant generic program libraries. These means are based on a sound and clear formal basis.
2. **Efficiency.** It implements this genericity in a highly efficient way regarding memory demands and execution time.
3. **Management.** It supports the management of source and object code by automatically maintaining code dependencies and avoiding code replication.
4. **Openness.** It is not a programming language itself but can be adopted to any programming language that implements the non-generic core constructions.
5. **Availability.** It is widely available, as well for users as for developers of generic libraries.

In the following subsections, we elaborate these objectives in somewhat more detail.

2.1 Genericity

There must be a *module language* that allows to formulate generic programs in a natural way such that (regarding genericity) the correctness of their implementation and of their application can be easily and independently verified. This module language is in charge of defining the interfaces between separate software entities and combining these entities to application programs ("programming in the large"). However, it is essentially independent of the *core language* that is in charge of implementing the internal details of these entities ("programming in the small").

In particular, the demands to the module language are as follows:

2.1.1 Generic Modules

It must be possible to write modules that can operate in the same way on objects of different types. Consequently, the programmer can reuse the previously developed source code when a new type of objects is introduced for which the corresponding functionality is required and makes sense.

> **Example.** A module implementing a sorting algorithm may be applied to multiple types of arrays. Likewise, the greatest common divisor of integers and of polynomials can be computed by the same algorithm.

The demand for generic source code does *not* necessarily imply that also the target code is generic; it is very well possible that for different types of objects different machine programs are generated.

Furthermore, while such a module may encapsulate some internal state, it is not necessarily itself a first order object i.e. it is not passed as a function argument or returned as a function result. This seems not necessary for the form of scientific/mathematical software that we aim at (in contrast to e.g. graphical user interfaces or client-server applications).

2.1.2 Specifications

A module may operate in the same way on objects of different types only if these types share a common behavior on which the module may rely. Generally the module has to demand from these types the existence of certain operations that fulfill certain mathematical laws.

> **Example.** A sorting algorithm for arrays demands from the element type a comparison operator $\leq$ that fulfills the laws of a total ordering. Likewise, the greatest common divisor can be computed only for objects that belong to a Euclidean domain.

In mathematics, the specification of such a collection of types and operations together with the laws these operations must fulfill is called an *algebra*. In programming languages a related concept is denoted by the term *specification*; we will stick to this term from now on.

All types of objects on which a generic module can operate thus represent instances of a common specification. Likewise the module itself represents a collection of types and operations that fulfill certain laws and therefore fulfills some specification.

We can therefore think of a generic module as a *parameterized module* that takes other modules as arguments. In order to give the parameterized module an independent meaning, the specifications of the module parameters must be part of its definition. Then the correctness of the implementation of the generic module can be checked by a compiler without referring to particular instances of its application. Vice versa, the correctness of its application can be controlled without referring to its implementation.

Furthermore, for a single module specification, several implementations may exist and be in use *in the same program*. Consequently, specifications should be independent entities to which different modules may refer.

In the spirit of algebras, these module specifications may not only describe the syntactic interfaces of operations but also their argument/result behavior. This behavior can be formally expressed by sentences in the language of *first-order logic*. Besides serving as a precise specification for the programmer, this preserves the information that may be utilized by a compiler to check the correct application of the module (as an argument to a generic module) and to verify its implementation.

In particular, the compiler may utilize the properties of functions (e.g. the distributive law) to apply optimizing code transformations. These specifications may also contain information about an operation's *strictness behavior* (i.e. which arguments may not be evaluated) or about the *linearity* of objects (i.e. whether an object is singly-referenced and may thus be destructively updated). In general, it should be possible to express in a specification any information that are sufficient and required for understanding the behavior of a function.

A module specification may also contain *default implementations* for some of the objects to be defined. These implementations can be only based on the information contained in the specification and is used provided that it is not superseded in a corresponding module implementation. Also formal specifications of operations might be used by the compiler to automatically generate default implementations.

In order to maximize the applicability of generic modules, the specification of a module shall be fully *independent of implementation issues* i.e. there should be a complete separation *what* properties a module must fulfill and *how* it actually fulfills these properties.

> **Example.** A module specification that demands the existence of a value a of type T it should be possible to implement this value by a variable $a : T$ as well as by a parameter-less function $a : () \to T$ without difference for the user.

2.1.3 Primitive Types and Type Constructors

The module language should be powerful enough to express the primitive types and operations provided by the core language. Consequently, it is possible to apply user-defined modules and predefined types without difference:

- On the one hand, it is allowed to pass predefined types as arguments to generic modules. Provided that the core language provides built-in integers, a generic sorting algorithm can operate on arrays of integers using the built-in integer operations.
- On the other hand, it is possible to provide new implementations for predefined types and operations. In a module that demands arbitrary precision arithmetic, a multi-word implementation of integers can replace the single-word implementation without difference for the user.

Like simple modules correspond to types, generic modules correspond to *type constructors*. Thus the module language should be powerful enough to express the type constructors built-in into the core language. Consequently, the built-in “array” constructor can in a certain context be replaced by a generic module without difference for the user.

2.1.4 Higher-Order Modules

The module system should make no fundamental difference between instantiated modules and generic (i.e. parameterized) modules. Consequently, we may parameterize modules by other generic modules i.e. by type constructors. It is then possible to replace all built-in constructors by corresponding user-defined generic modules.

Correspondingly, we want the possibility to let a generic module return another generic module by *currying*: in a curried application of a generic module, actual arguments are provided only for a subset of the formal parameters yielding a generic module with the remaining parameters still to be bound.

> **Example.** The generic module (i.e. type constructor) ‘array2(I,E)’ is parameterized by the index type I and by the element type E. The curried application ‘array2(int,_)’ then yields an array constructor ‘array(E)’ with integer indices. A generic library may thus utilize different levels of flexibility for different purposes.

2.1.5 Importing and Overloading

It should be possible to refer to the components of a module either by explicit qualification by the module name or by *importing* all bindings defined in another module into the current module. Besides reducing the writing overhead, it is then possible to use simple operator syntax for generic operations (e.g. $x + y$ for all ring operations).

Moreover, the *overloading* of functions/operators i.e. the use of the same name for different operations should be supported. The actual function can be automatically selected by considering the types of the operands. Consequently, it is possible to use in a single module the operator + simultaneously for *different* ring operations, e.g. for integer arithmetic and for rational arithmetic. The user must also be able to uniquely specify which operation is denoted by an overloaded operator (by explicit qualification with argument/result types).

An issue that arises in this context is also the support of overloaded literals since a phrase like '`123`' does not necessarily denote an object of a predefined type any more.

2.1.6 Inheritance

It should be possible to derive in a simple way new module specifications by inheritance of existing specifications. The derived specification extends the contents of the base specification by additional operations and/or laws that any corresponding module must fulfill. Also multiple inheritance of several sources should be supported; therefore mechanisms for resolving name clashes have to be provided.

Correspondingly, it should be possible to derive in a simple way new module implementations by inheritance of existing implementations. The derived module may extend and/or supersede the definitions of the base module.

2.2 Efficiency

Since we aim at the development of *high-performance* generic software, there are several requirements for the implementation of genericity. In a nutshell, we do not want the abstraction provided by generic programming to be penalized by additional runtime and/or memory overhead. Otherwise for performance-bound applications, programmers would be forced to abandon the use of genericity. Consequently our demands are the following:

2.2.1 Runtime Efficiency

Using generic modules must not be penalized by additional execution overhead. There must be no runtime difference between the application of a function that is implemented in a non-generic software system and the application of a corresponding function that is defined in a generic module or in a module on which a generic module depends.

This constraint is extremely strict but necessary to exploit genericity at any level of the module hierarchy of a complex software package without substantial loss of efficiency.

> **Example.** Take an implementation technique that adds a single function call to every application of an operation defined in an argument module: if the whole application is parameterized over the integer numbers, adding a 5-clock function call to every 1-clock integer operation may multiply the execution time of the program by a factor of 6.

2.2.2 Representation Efficiency

Using generic modules must not be penalized by additional representation overhead; there must be no additional constraints for the memory representation of a runtime value that is defined in a generic module or in a module on which a generic module depends compared to the memory representation of a corresponding value implemented in a non-generic software system.

Again, this restriction is very strict but nevertheless necessary to exploit genericity at any level of the software hierarchy without a dramatic increase of its memory requirements.

Example. Take an implementation that demands that every object defined in an argument module must be implemented as a pointer to a heap structure: instead of a single word floating point number, this technique requires a single word pointer plus the floating point number plus additional heap management information (say one word). Besides multiplying the memory demands by a factor of 3, every access to a floating point number is then also penalized by an additional dereferencing operation, which again increases the runtime of the program.

2.3 Management

Building and using generic software libraries also demands assistance for the management of the generated code; otherwise the complexity of software maintenance would soon become overwhelming and make an effective use difficult. These issues do less affect the formal layer of the module language but more the supporting software environment:

2.3.1 Namespaces

It must be possible to restrict the namespace of module names. The motivation is to avoid name clashes between modules that are defined in different libraries but used within a single application. Thus sets of modules can be grouped into a package such that all modules within a package share the same namespace but the use of all other modules needs qualification by the package name (with possibilities for "importing" the contents of namespaces).

2.3.2 Management of Code Dependencies

The development of libraries and application programs requires the management of various kinds of source code (specifications and modules) and object code (compiled modules, libraries/archives, executable programs). In a generic environment, this organization becomes considerably more complex by the existence of generic code that can be instantiated multiple times in multiple ways (provided that different instantiations generate different object code files).

It is necessary that the programming environment frees the programmer from the task of maintaining the code consistency by automatically tracking dependencies, recompiling source code, updating libraries and relinking executables, if they are affected by modifications of the source code. It is also highly desirable, that the management of multiple source code versions is supported.

2.3.3 Sharing of Libraries

It is common practice to have in a computer system a single copy of every programming library that can be shared by all application programmers. Such libraries typically consist of interface specifications (for checking the correct use of the entities in the library) and of compiled object code (for linking with the custom code of the application). The compiler and the linker (and the loader in case of dynamic link libraries) use these components automatically with only minor interference by the programmer.

For libraries that are based on generic modules, the corresponding concept should hold with a crucial generalization: if the system generates code not only for every generic module but for every *instantiation* of a generic module, also the code derived from the instantiations should be automatically shared among all programmers without manual interference. Otherwise, the disk space requirements of the system would soon become prohibitive. Likewise, within a program code should not be replicated for identical instances of a generic module created in different modules.

2.3.4 Standard Linking Interface

A pragmatic issue that is crucial for the practical application of a generic programming environment is its embedding into a conventional (say Unix) environment. This means on the one hand the possibility to call functions implemented in other languages (in particular C and C++). However, even more important is the ability to link the generated code with application programs implemented in other languages. Therefore the generic programming system should ultimately generate from a (fully instantiated) module a Unix library with a standard calling and linking interface.

2.4 Openness

The system should be orthogonal with respect to the core language in which the small scale entities are actually defined.

There are two reasons for this demand:

- We would like to keep the module language independent of syntactic, semantic, or implementation peculiarities that stem from a particular core language. Especially, the module language shall *not* be bound to a particular (e.g. functional) programming paradigm.
- We would like to maintain the possibility to bind the module language to different core languages (that programmers in different application areas are familiar with) or use the module system for the development of new languages. The implementation of this module language must therefore be itself *generic* with respect to the core language.

Of course, the module language should have a predefined binding to some wide-spread core language (e.g. C). However, it should be possible to provide bindings for other existing or future languages as well. This demands a flexible and well-structured implementation and clear interface specifications based on a *formal semantics*.

2.5 Availability

A remaining issue to be considered is not a technical but a political one. Our ultimate goal is to develop software libraries that can be distributed among many users in various scientific domains for free use in their own applications. "Using" in the context of generic software means not only linking precompiled object code, but also

1. generating new instances of generic modules, and
2. building own modules based on previously defined modules.

This requires the availability of the generic programming environment to application programmers (instantiating existing generic libraries) and library developers (building new libraries).

In order to maximize the possibilities for distributing the library, the programming environment must be either very wide spread among the interested community or better *freely available* for any interested party (e.g. via file servers on the Internet). In other words, we do not want to severely restrict the distribution of the developed software because it is based on a proprietary system that is not in general use and must be first commercially purchased by interested parties.

Chapter 3

Concepts

In this chapter, we sketch various concepts that have been suggested for generic programming and justify the choice of functors as the base of the proposed project. Unfortunately the terminology used in literature is not consistent such that sometimes different names are used for similar concepts or, worse, the same name is overloaded with different concepts. In the following, we will try to clarify the situation by elaborating four major approaches that have been proposed in the past:

1. **Genericity by Polymorphism.** A polymorphic function may operate on objects of unknown type. Genericity may be expressed by passing all required operations as additional function arguments.
2. **Genericity by Preprocessing.** A generic module refers to abstract type and function names; an instantiated module is created by textually replacing these names with the names of concrete types and functions.
3. **Genericity by Inheritance.** A generic function operates on objects whose types inherit from the same base class; all objects carry the actual instances of the class methods.
4. **Genericity by Functors.** A generic module is a "functor" i.e. a function that maps argument modules into a result module.

3.1 Genericity by Polymorphism

The concept of parametric polymorphism is based on the theory of *second-order typed lambda calculus* [Cardelli and Wegner, 1985, Reynolds, 1990, Huet, 1990, Schmidt, 1994] for an introduction. It augments the type system with *type variables* such that one may write functions like

```
length: List(*a) -> Int
length(l) = if isnil(l)
            then 0
            else 1+length(tail(l))
```

(where $*a$ is a type variable) that operates in the same way on lists of all types of elements. Since the function does not know the element type, it must not apply any operations to it. It is therefore not possible to write functions like

```
eqlist: (List(*a), List(*a)) -> Bool
eqlist(l, r) = if isnil(l)
               then if isnil(r) then true else false
```

```
                    else if head(l) == head(r)
                          then true
                          else eqlist(tail(l), tail(r))
```

where an operation `==: (*a,*a) -> Bool` is required. A solution is to pass this function as an additional parameter to the generic function i.e.

```
eqlist: (List(*a), List(*a), (*a,*a) -> Bool) -> Bool
eqlist(l, r, eq) = ... eq(head(l), head(r)) ...
```

Of course, this is only possible in a *higher-order language* where functions may be passed as arguments to other functions.

If several functions are required, we may pack them into a single record and pass the record as an additional argument. Such a record thus represents a *module* that exports all the required functions. For instance, the record

```
Integer: Ring(Int) = record { sum = intsum, ... }
```

is a module whose type `Ring(Int)` is an instance of the polymorphic type

```
Ring(*a) = Record {
  sum: (*a, *a) -> *a
  ...
}
```

A generic module is thus a *function* that takes modules as arguments and returns a module as result, e.g.

```
PolyRing: Ring(*a) -> Ring(Poly(*a))
PolyRing(R) =
  let
    polysum: (Poly(*a), Poly(*a)) -> Poly(*a)
    polysum(a, b) = ... R.sum(first(a), first(b)) ...
    ...
  in
    record { sum = polysum, ... }
```

This approach is theoretically appealing and can be principally used in many languages, as well in higher-order languages with static types (ML [Milner *et al.*, 1990] or Haskell [Hudak *et al.*, 1992]) as in languages with dynamic types (e.g. Lisp [McCarthy, 1960, Steele Jr., 1984] and Mathematica [Wolfram, 1991]) and even in low-level imperative languages (e.g. C [Kernighan and Ritchie, 1988]).

However, this concept has also severe semantic and pragmatic deficiencies:

1. In a language with a simple static type system, a record does not adequately model the encapsulation aspect of a module because components cannot be "hidden". The record type reflects all components and therefore does actually not describe the specification but the *implementation* of the module.

   We therefore either have to give up static type checking or use a language that supports parametric polymorphism in combination with a notion of *subtyping* [Cardelli and Wegner, 1985, Weber, 1993] where an extended record is type-compatible with the base record. However, as described in Section 3.3, several semantic problems arise in such languages as well.

2. Records are run-time entities; access to a module component requires a record lookup at runtime. Furthermore, all generic operation must be true functions; primitive operations have to be encapsulated by user-defined functions. Both issues make this approach rather inefficient.

3. All objects must have representations of the same size (say one word). Any larger object (e.g. a double word floating point number) must be *boxed* by allocating it on the heap and using a (one word) pointer instead. This again increases the memory and runtime overhead.

All these reasons make this approach less appropriate for large-scale generic software development.

3.2 Genericity by Preprocessing

Genericity can be also implemented by a preprocessor for the textual replacement of type and/or function names. For instance, a source file defining a set of vector operations may contain the definition

```
#include <<T>.h>
...
<T>Vector <T>VecScalarProd(<T>Vector v, <T> c) {
  ... <T>Prod(v[i], c) ...
}
```

A preprocessor may be applied to replace `<T>` with `Int` and generate from the generic source file an instantiated file

```
#include <Int.h>
...
IntVector IntVecScalarProd(IntVector v, Int c) {
  ... IntProd(v[i], c) ...
}
```

that can be compiled and linked to the application program as usual. It is the responsibility of the programmer to provide a header file `Int.h` (which contains the definition of type `Int` and the interfaces of the associated operations such as `IntProd`) and to link the program with a corresponding object file.

This strategy can be implemented in a rather straight-forward way in C [Kernighan and Ritchie, 1988] using the facilities of the standard preprocessor.

> At RISC-Linz, the generic computer algebra library GRÖBNER [Windsteiger and Buchberger, 1993, Windsteiger, 1993] was developed for computing Gröbner Bases. The library was implemented in C and used the C preprocessor for compile-time instantiation of the source code with different mathematical domains.

This is also possible in C++ [Stroustrup, 1991, Stroustrup, 1994] where it is by overloading not even necessary to invent artificial function names. Instead one may simply write

```
#include <<T>.h>
...
<T>Vector operator Prod(<T>Vector v, <T> c) {
  ... Prod(v[i], c) ...
}
```

using the same function name `Prod` for the addition of vectors and of vector elements. The functions become even more elegant, if the C++ operator syntax is used. This approach actually was pursued by several C++ computers before *templates* were introduced [Stallman, 1992].

The main problem with this approach is that it is not the reference to a particular instance of a generic module that triggers the generation of code for this instance. Instead it is the responsibility of the programmer to keep track of all instances of generic modules used in a program and to generate the appropriate instantiated code.

Also the *template* construct of C++ is just such a textual replacement mechanism where however the compiler itself generates the required instances and frees the programmer of this tedious management task. The programmer may thus write a generic class

```
template <class T>
  class Vector {
    ...
    Vector sum(Vector v, T c) {
      ... Prod(v[i], c) ...
    }
  }
```

When the C++ compiler detects an application

```
#include <Vector.h>
#include <Int.h>
...
Vector<Int>
```

the template will be automatically instantiated and compiled. The C++ language definition demands that code is not replicated when a template is used with the same template arguments in different translation units (still code is replicated for different users of a template library). However, some C++ compilers (most notably the GNU compiler g++) deliberately ignore this requirement and replicate code; the programmer may optionally suppressed code generation but is then again left with the task of managing template instantiations.

In any case a crucial deficiency remains with the template approach of C++: since no information is available about the template arguments, it is *not* possible to compile and type-check a template on its own and to check the correctness of its use by just referring to its interface. Instead, all type checking is postponed to template instantiation time. Stroustrup argues in [Stroustrup, 1988]

> Should a user be required to specify the set of operations that may be used for a template argument of type *type*? ... No. Requiring the user to provide such information decreases the flexibility of the parameterization facility without easing the implementation or increasing the safety of the facility. ... It has been argued that it is easier to read and understand parameterized types when the full set of operations on a type parameter is specified. I see two problems with this: such lists would often be long enough to be unreadable and a higher number of templates would be needed for may applications.

Later [Stroustrup, 1994] Stroustrup actually concedes the importance of constraints in readability and early error detection and describes two alternatives for expressing them (through inheritance from a base class and through use of the required operations in a dummy function). However, both alternatives are poor substitutes that still do not solve the basic problem: a template has to be compiled and type-checked *repeatedly* for *every* application; it has no independent semantics of its own. This deficiency makes it difficult to heavily rely on templates for large scale generic software development.

3.3 Genericity by Inheritance

3.3.1 General

In object-oriented programming languages, a function may operate on objects of different classes provided that these classes inherit (i.e. are derived) from the same base class. Consequently, using the class

```
Ring = class {
  method +(Ring): Ring
  ...
}
```

a generic function

```
fun sumprod(x: Ring, y: Ring, z: Ring): Ring =
  x.+(y.*(z)) /* x+y*z */
```

can be defined that may operate on arguments of *any* subclass of `Ring`.

Actually, the class `Ring` just serves as the interface of its subclasses; it need not itself provide definitions for the denoted methods. In the terminology of C++, `Ring` is a *virtual class*: objects cannot be created from such a class itself but only from its subclasses which provide all required method definitions.

The implementation of virtual classes is based on function tables that are looked up at runtime. This implementation technique on the one hand yields some run-time overhead (that may soon become prohibitive if applied in several layers) and also demands every operation to be represented as a function (without possibility of inlining primitive operations).

3.3.2 Inheritance and Subtyping

However, neglecting performance issues such a subclass should be for our purpose

```
Int = class(Ring) {
  var value: int
  method +(x: Int): Int = ... value + x.value ...
  ...
}
```

that describes the internal structure of an `Int` object and the actual definition of the `Ring` methods.

Unfortunately, in most object-oriented languages this definition of `Int` is *not* possible because the type interface of class methods must not be changed in subclasses. It is thus only allowed to write a class

```
Int1 = class(Ring) {
  int value;
  method +(x: Ring): Ring = ...
  ...
}
```

that does not change the type of `+`. This restriction was introduced to guarantee type safety when a function argument is from a subclass of the parameter class. Without this restriction, definitions like

```
Nat = class {
  var value: int;
  method eq(x: Nat): Bool = (value == x.value)
  ...
}

Int2 = class(Nat) {
  var sign: bool;
  method eq(x: Int2): Bool = (sign == x.sign) & (value == x.value)
  ...
}
```

would be allowed. Then it would be also possible to write functions as

```
fun f(n1: Nat, n2: Nat) = n1.eq(n2)
fun g(i1: Int2, n2: Nat) = f(i1, n2)
```

where the application of `f` is valid since `Int2` is a subclass of `Nat`. But then the application `n1.eq(n2)` yields a run-time error, since `n1.eq` refers to the method in class `Int2` which actually expects an `Int2` (not a `Nat`) argument.

Above example illustrates a discrepancy in the type system of object-oriented languages that has been summarized by the phrase "subclasses are not subtypes" [Cook *et al.*, 1990]. This discrepancy results from the combination of method redefinition in subclasses and the use of objects from subclasses as method parameters.

For this reason, in most statically typed object oriented languages (C++ [Stroustrup, 1991], Simula [Birtwistle *et al.*, 1979], Modula-3 [Nelson, 1991]) the types of methods must not be changed in subclasses. Eiffel [Meyer, 1988, Meyer, 1991] provides class inheritance where one may write essentially a definition like

```
Ring = class {
  method +(Current): Current
  ...
}
```

where `Current` refers to the current class (i.e. the class into which it is inherited, e.g. `Int`). However, its type system is subject to the anomaly described above and thus inherently insecure.

Formal models have been proposed that formalize a notion of subtyping under which it is safe to pass objects from subclasses as method arguments; the resulting typing systems are restrictive but safe [Cook *et al.*, 1990, Gunter and Mitchell, 1993].

[Baumgartner and Russo, 1994b, Baumgartner and Russo, 1994a] propose to completely separate in C++ inheritance (the reuse of code of existing classes) from subtyping (defining the relationships between classes) by the introduction of (ML-like) signatures; a prototype implementation has been incorporated into the GNU C++ compiler.

3.3.3 Methods and Functions

Let us for the moment assume that we have such a type system where above definition of `Int` based on `Ring` is allowed. Then the programmer may outside the definition of `Ring` write above generic function `sumprod` that operates on *any* `Ring` objects. However, an application `sumprod(a,b,c)` to `Int` objects `a`, `b`, `c` is still prohibited!

The problem is that the ring operation `a.+` expects an argument of type `Int` but, in the context of the function `sumprod`, we only know the argument of `x.+` to be of type `Ring`. This problem could only be solved, if in the definition of `sumprod` we were able to specify that `a`, `b` and `c` are objects from the *same* subclass of `Ring`.

Thus either the compiler refuses the function definition or also mixed applications like `sumprod(a,b,d)` are allowed where `d` is an object from a *different* subclass of `Ring`; then the resulting type system is insecure.

The only clean solution is to make also `sumprod` a method of class `Ring`

```
Ring = class {
  method +(Ring): Ring
  method sumprod(Ring, Ring): Ring
  ...
}
```

as well as *any* other generic function that we would like to provide. On the one hand, this yields a rather tedious programming style since it only generic class methods (but not independent generic functions) are allowed. On the other hand, genericity is only supported with respect to a single class; a generic method cannot depend on multiple class arguments.

Summarizing, the type systems of most available object-oriented languages are either not safe or not powerful enough for expressing genericity by inheritance from virtual base classes. There exist typing systems where genericity can be safely expressed but the results are still not really satisfying for our purposes.

3.4 Genericity by Functors

3.4.1 General

Several programming languages allow a collection of exported entities (types, functions, values) to be parameterized by other such collections. This mechanism is e.g. called *generic package* (Ada [Standard, 1983, Standard, 1994]), *parameterized domain* (A# [Watt *et al.*, 1994], Axiom [Jenks and Sutor, 1992], Scratchpad [Watt *et al.*, 1988]), or *functor* (ML [Milner, 1984, MacQueen, 1984, Milner *et al.*, 1990]). After instantiation of the arguments, the result is a *package* (Ada), *domain* (A#), or *structure* (ML). In this section, we will generally stick to the ML terminology.

This approach differs from "genericity by preprocessing" (Subsection 3.2) in that every structure has a specific "metatype" describing the interface of all exported entities. These metatypes are called *generic parameter declarations* (Ada), *categories* (A#), or *signatures* (ML). In a functor definition, the signatures of the parameters are explicitly given such that the functor can be compiled and type checked on its own. For every instantiation of the function, only the signatures of the actual argument structures and the formal functor parameters have to be matched to ensure correctness of the instantiation.

Take e.g. the signature

```
Ring = signature {
  type T
  fun +(T,T): T
  ...
}
```

which is matched by the structure

```
Int = struct {
  type T = int
  fun +(T,T): T = ...
  ...
}
```

In a functor definition

```
Gen = functor(R:Ring) {
  + = R.+, * = R.*, T = R.T
  sumprod(x:T, y:T, z:T) = x+y*z
  ...
}
```

the generic function `sumprod` can be defined for *any* `Ring` type. The instantiation

```
I = Gen(Int)
```

generates a structure whose component `I.sumprod` operates on `Int` objects.

3.4.2 Specifications

There are different ways how signatures and structures/functors are associated. In Ada, signatures are not separate entities but every generic parameter declaration has to list all components separately; actually a package cannot be parameterized by other packages but only by individual components (types, functions, values) that have to be separately passed in every instantiation.

In ML and A#, signatures/categories can be separate entities that can be freely associated to structures, functor/domain arguments and results. A# classifies its domains as *abstract datatypes* (defining a distinguished type) or *packages* (providing a collection of exported objects including types). An A# domain that qualifies as an abstract datatype can be used in any place where a built-in type may occur. ML structures are essentially equivalent to A# packages. C++ classes are similar to A# abstract datatypes.

3.4.3 Inheritance

Both ML and A# provide means to extend signatures and structures by inheritance from other signatures and structures. In ML, the `include S` directive adds all bindings from signature `S` to the current signature. Likewise, the `open S` directive adds all bindings from structure `S` to the current structure. A# allows to extend categories with the declaration `C1 with C2` that creates a category joining all bindings in `C1` with those in `C2`. Correspondingly, the declaration `D1 add D2` creates a domain consisting of all the bindings in `D1` joined with the bindings in `D2`. Moreover, A# supports the "post facto extension" of domains by a definition `extend D1:C = D2` that adds all bindings in `D2` to the previously defined domain `D1`.

3.4.4 Qualification and Import

The environment provided by an ML structure may be imported into the current environment by a directive `open S`. However, it is also possible to explicitly refer to the exported component `c` of a structure `S` by indexing `S.c`. This is also true if the exported component is an abstract datatype whose representation is hidden but that may be used like any other type. This form is also sometimes called *strong abstract type*. On the contrary, A# only supports only the import of whole environments of an abstract datatype which is called *weak abstract types*.

With the strong form of abstract datatypes, it is rather easy to build in a module-based system an abstract type and distribute this type to multiple modules of a system that may build upon this type and still interface correctly. Also different instances of the same abstract type may be easily mixed within the same program; by the encapsulation of module environments, interference can be easily avoided. However, with the weak form of abstract datatypes, it is very difficult to deal with such situations [Schmidt, 1994].

3.4.5 Sharing

Sometimes, it is necessary to check that different abstract types passed as functor arguments refer to the same concrete representation but without "giving away" the representation. Take for instance a ML functor, where two argument structures refer to the abstract type `Int`. For combined use, it is necessary to verify that both versions of `Int` refer to the same representation. In ML, this is possible by a signature clause `sharing S1 = S2` that demands that both structures `S1` and `S2` are identical:

```
F = functor(A: S, B: S sharing A.Int = B.Int) {
  ...
}
```

In A#, conditional expressions may be used to test whether a domain passed as a functor argument satisfies additional categories beyond that denoted in the parameter specification. By these conditional expressions, as well executable code may be conditionalized as well as declarations and definitions. However, A# does not (yet) support the test for the equality of domains, which would be the equivalent of the ML `sharing` clause.

3.4.6 Axiomatic Specifications

In order to develop a formal framework for program development, [Sannella and Tarlecki, 1991, Kahrs *et al.*, 1993, Kahrs *et al.*, 1994] introduce Extended ML (EML). EML extends SML signatures to *specifications* by adding *axioms* (logic formulas with universal and existential quantification) that any structure matching this specification must fulfill. A formal verification model of EML has been developed that allows the programmer to derive by a sequence of proven transformation steps a correct SML program from an EML specification. EML is therefore not a programming language itself (there exist no EML compilers) but the result of the transformation process is an executable SML program.

3.4.7 Higher Order Functors

In Ada and in the original module system of Standard ML (SML), functors are first order i.e. parameters and results can be only modules and functors could not be components of modules.

Later analyses of the SML module system [MacQueen, 1986, Mitchell and Harper, 1988, Harper *et al.*, 1990] made it clear that it was natural to extend the notion of functors to higher order by allowing functors as parameters and results (or, equivalently, as components of modules). Besides just making the language more symmetrical, this extension supports new modes of modularization that are useful and important for the development of software libraries [Berry, 1993].

> One consequence of higher-order functors is the possibility of *functor currying* i.e. the partial instantiation of functors. This allows to use the most generic form of a functor (say with two arguments) as the basis of a more concrete form (with one argument) required in a particular context.

While the conceptual extension is natural and poses few theoretical difficulties, it becomes challenging in the context of SML: here signature matching is "transparent" i.e. the identity of types is propagated through functor calls such that sharing information may be automatically derived from functor implementations (this is in contrast to opaque signature matching where the only sharing information is derived from signatures). These technical difficulties were mastered in [Tofte, 1992, MacQueen and Tofte, 1994, Harper and Lillibridge, 1994] and a prototype implementations of higher-order functors was recently undertaken [Cregut and MacQueen, 1994].

3.4.8 Implementation

For the implementation of the functor language, Ada, ML, and A# use different approaches:

1. In Ada, for every application of a generic module, instantiated code is generated (corresponding to the implementation of C++ templates). Generic packages are therefore pure compile-time concepts.

2. In present ML implementations, code is generated for every functor before the arguments are actually present. At runtime, functor application is then ordinary function application (with structures as arguments and as results). Semantically, however, functors are a higher-order concept that cannot be mixed with functions. This separation between the *module language* and the *value language* was introduced to guarantee compile-time type checking of ML programs in the presence of strong abstract types [MacQueen, 1984, Schmidt, 1994].
3. In A#, like in ML, code is generated for every functor. However, unlike in ML, A# functors are not a separate concepts but ordinary functions that receive domains as arguments and return domains as result. Compile-time type checking is still possible since A# only supports the weak form of abstract datatypes.

From the performance point of view, only Ada implements generic packages without additional runtime and/or memory overhead. In ML and A#, structures/domains are essentially records and functors are just functions over these records. Consequently, every reference to an abstract datatype operation requires (similar to object-oriented languages) a table lookup at runtime and all concrete representations of abstract datatypes have to be of the same size (i.e. they are usually pointers to heap objects).

The resulting performance degradation is only acceptable, because apparently most ML and A# programs make only limited use of functors. Most operations directly refer to concrete (i.e. built-in) types that can be resolved by the compiler. Furthermore, much interaction between modules seems not to rely on functor parameters but on direct "imports" of concrete module implementations. While this helps to restrict the performance overhead, it also limits the reusability of programs.

3.5 Summary

Genericity by polymorphism is only adequate in dynamically typed programming languages whose performance however is to poor for large-scale applications. Genericity by preprocessing is efficient but crude because it lacks the sound interfaces required for programming in the large.

Object-oriented languages provide genericity by inheritance from virtual base classes. This concept is implemented by table lookup at run-time with the associated restrictions and performance penalties. Furthermore, the inheritance mechanism does not adequately model the subtyping concept required for expressing mathematical domains.

All in all, the functor-based programming style is the most promising one for expressing genericity in a language with a static type system. In particular it may not only be implemented by table lookup at runtime but also by code instantiation at compile-time without performance degradation.

Chapter 4

State of the Art

4.1 General

The functor-based approach described in the Chapter 3 has been implemented in various programming languages and systems in various ways and to various extent. In the following, we will evaluate some of these languages and compare their features against the requirements stated in Chapter 2:

1. Ada [Standard, 1983, Standard, 1994]
2. C++ [Stroustrup, 1991, Stroustrup, 1994]
3. Standard ML [Milner, 1984, Milner *et al.*, 1990]
4. A# [Watt *et al.*, 1994]
5. Haskell [Hudak *et al.*, 1992]

This enumeration contains those languages that seem to be the most promising candidates for writing generic programs. It is by no means exhaustive but still should give a representative cross-section of the current state of the art. We have deliberately omitted many languages, in particular languages

- that are not statically typed, because such languages are too inefficient for high-performance programming (e.g. Mathematica [Wolfram, 1991]),
- that only provide inheritance to express genericity, because this concept is too weak for our purpose (object-oriented languages, e.g. Eiffel [Meyer, 1988, Meyer, 1991]),
- whose generic features are not fundamentally different from some of the listed languages (e.g. Modula-3 [Nelson, 1991]),
- that are primarily intended for research in type systems and not for large-scale software development (e.g. OBJ [Futatsugi *et al.*, 1985, Goguen and Winkler, 1988]).

On the other hand, we include C++ because its templates bear sufficient similarities to functors (while actually being just a preprocessing mechanism) and because of the widespread success of the language.

One might argue that some of our evaluation criteria are not adequate since they do not refer to the *definition* of a programming language but just to its *implementation*. Of course we concede that a language may be implemented in various ways with different characteristics. Practice however shows that languages are generally designed with a certain implementation technique in mind which makes it rather difficult to develop (without changes in the language

	Gen	**Spe**	**Par**	**For**	**Def**	**Abs**	**Typ**	**Con**	**Hig**	**Qua**
Ada	++	0	0	−	−	+	0	−	−	++
C++	++	−	−	−	−	+	+	0	−	++
ML	++	++	++	−	−	+	−	−	+	++
A#	+	++	+	−	++	+	++	0	++	−
Haskell	+	+	+	−	++	+	++	−	−	−

Gen Generic modules.
Spe Separate module specifications.
Par Specifications of module parameters.
For Formal specification of module components.
Def Default implementations in specifications.
Abs Abstraction of specifications from implementations.
Typ Power to cope without predefined types.
Con Power to cope without predefined type constructors.
Hig Higher-order modules and currying.
Qua Qualification of modules.

	Imp	**Ove**	**Ins**	**Ini**	**Run**	**Rep**	**Nam**	**Man**	**Sha**	**Lin**
Ada	++	++	−	−	++	++	0	++	−	++
C++	−	++	−	++	++	++	+	++	−	++
ML	++	−	++	++	−	−	−	0	␣	0
A#	++	++	++	++	−	−	−	0	␣	++
Haskell	++	++	++	+	−	−	−	0	␣	0

Imp Importing of modules.
Ove Overloading of functions/operations/constants.
Ins Multiple inheritance of specifications.
Ini Multiple inheritance of classes.
Run Runtime efficiency.
Rep Representation efficiency.
Nam Namespaces.
Man Management of code dependencies.
Sha Sharing of instantiation code in libraries.
Lin Standard linking interface.

	Ope	**Sem**	**Ava**
Ada	−	+	+
C++	−	−	++
ML	0	++	+
A#	−	−	0
Haskell	−	++	+

Ope Openness to different core languages.
Sem Formal semantics.
Ava Availability to software users and developers.

Figure 4.1: Evaluation

definition) a fundamentally different implementation. This can be seen in languages such as Ada and C++ that are industrially supported with many independent implementations.

Figure 4.1 summarizes the results of our evaluation in a table where we estimate to which the degree each language meets each of the objectives summarized in Chapter 2. The grade assignments are − (no support), 0 (some support), + (good support), ++ (optimum support), ␣ (not relevant). Of course, these grades are just coarse estimations and should be taken *cum grano salis*. In total, however, we feel that they give a representative picture for the applicability of the language for high-performance generic programming.

4.2 Ada

Ada [Standard, 1983, Barnes, 1994, Bruce, 1994] has been developed in the late 1970s on behalf of the US Department of Defense (DoD) as an attempt to provide a universal programming language for implementing mission-critical systems. At the beginning of the 1990s a major revision of the language was devised to include object-oriented features (i.e. inheritance) [Taft, 1992] and recently standardized as Ada 95 [Standard, 1994, Naiditch, 1995]. However, apart from the supplying industry for US governmental institutions, the language failed to gain wide-spread acceptance, especially in Europe.

Ada provides generic packages and generic procedures (Gen ++). A package consists of a package specification and a separate package body, but it is not possible to use the same specification for different bodies in the same context (Spe 0). Generic packages may be parameterized by types, values, functions and procedures but not by other packages; parameter interfaces are contained in the generic package specification without the possibility of specifying type sharing (Par 0).

As a substitute for the missing possibility of passing packages as arguments, parameter types may be constrained to specific categories (e.g. numeric types or array types); certain operations are then assumed on these types. Furthermore, argument functions may be omitted; in this case either default implementations are used or those functions are used that are named like the formal parameter and are visible at the point of instantiation (the interfaces of argument functions are individually specified in the generic header).

It is not possible to formally specify the behavior of functions (For −) or to provide default implementations of exported items (Def −). Package specifications are fairly abstract, but it is e.g. not allowed to implement a constant by a parameter-less function (Abs +). It is not possible to replace all occurrences of pre-defined types by user-defined types, in particular not (numeric) loop indices or (boolean) loop conditions (Typ 0).

By generic modules, it is possible to define an own array type (constructor) but its use differs considerably from the use of the standard constructor and is more limited (since the compiler does not recognize the generated types as array types). It is essentially not possible to define replacements of the other constructors (Con −). Higher-order generic classes are not supported (Hig −). It is possible to access exported items by package qualification (Qua ++) or to import the contents of another package into the current package (Imp ++); overloading is appropriately resolved (Ove ++).

A package specification may not directly inherit other specifications (Ins −) and package bodies may not directly inherit other bodies (Ini −). Instantiations of generic parameters are performed at compile-time; runtime and representation efficiency are therefore optimal (Run ++, Rep ++). There is no support to restrict the namespace of package names but package bodies (not specifications) may be separately compiled as subunits of other packages (Nam 0).

Ada environments provide good support for the dependence control of files (Man ++) but no support for the sharing of instantiated generic code among different users (Sha −).

Generally a standard linking interface is applied (Lin ++). The module system is tightly bound to the programming language (Ope −). Ada is principally widely available but must be separately purchased; a free variant has recently become available under the GNU copyleft (Ava +).

Ada is a large language that has been thoroughly specified in natural language [Standard, 1983]; attempts at formal definitions have been pursued [Kahn and others, 1980] (Sem +). Compilers must pass a validation suite to be entitled to wear the official Ada label.

4.3 C++

C++ [Stroustrup, 1991, Stroustrup, 1994] was developed around 1985 by Bjarne Stroustrup as an attempt to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming. The language has been an extremely large success and is widely supported by industry and academics. ANSI/ISO standards for C++ and its class library are pending.

C++ provides two solutions for expressing genericity: virtual classes and class templates. However, as explained in Subsection 3.3, the former concept is too weak and too inefficient for our purposes. In the following description, we will therefore concentrate on the later.

C++ templates are similar to Ada generic packages in that classes may be parameterized by types and compile-time constants (Gen ++). However, there is no separate specification of a template interface but the whole template implementation must be known at the point of use (Spe −). Also it is not possible to constrain template arguments; each template instantiation must consequently be compiled and type-checked on its own (Par −).

It is not possible to formally specify the behavior of operations (Spec −) or to give default implementations of functions (Def −). Class interfaces are fairly abstract, but it is e.g. not possible to implement a constant by a parameter-less function (Abs +).

Of all languages mentioned in this proposal, only C++ supports the automatic conversion of a value into a value of another type, if this value is argument to a function and its type does not match the type of the formal parameter. Consequently, it is possible to use user-defined types as replacements of all builtin types. However, builtin types are themselves not classes which lets them not appear in all contexts that classes may appear (Typ +). It is possible to replace arrays and pointers (but not other builtin type constructors) by own variants (Con 0). Higher-order templates are not supported (Hig −).

While it is possible to import the contents of namespaces (see below) into the current scopes, this is not true for template classes (Imp −). Overloading is fully supported (Ove ++). Template specifications do not exist and may thus not be inherited (Ins −) but classes may inherit other classes (Ini ++). Since code is generated for every instantiation, runtime and representation efficiency are optimal (Run ++, Rep ++).

The ANSI standard introduces namespaces to restrict the scope of class names but not all compilers support this concept yet (Nam +). C++ environments provide good support for code management (Man ++) but sharing of instantiated template code among different users is generally not possible. Worse, despite of the language definition some C++ compilers generate duplicated code for identical template instantiations in different object files (Sha −). Standard linking interfaces are supported (Lin ++). The template mechanism is tightly bound to the programming language (Ope −).

C++ is a very complex language with many features whose interaction is sometimes not really well understood. While the language is specified in a draft ANSI/ISO standard, no attempt of a formal definition has been (and probably will be) ever made (Sem −). The language is widely available also with free implementations (Ava ++) but because of the complexity of the language different C++ compilers sometimes disagree on subtle features.

[Caprotti *et al.*, 1994] describes how a generic library for algebraic computation is implemented in C++. The main decision is to use C++ templates, since virtual functions are too inefficient for low level operations like integer arithmetic. The template mechanism unfortunately is rather poor to support fully the genericity involved in the basic classes of the library. Also the pragmatic issues and problems are discussed that arise when generic C++ code has to be managed among multiple users. [Akerkar *et al.*, 1994] elaborates why the lack of namespaces in some current C++ compilers cannot be really overcome by use of nested classes.

[Baumgartner and Russo, 1994b, Baumgartner and Russo, 1994a] propose to separate inheritance (the reuse of code of existing classes) from subtyping (defining the relationships between classes) by the introduction of signatures (in the sense of ML); a prototype implementation has been incorporated into the GNU C++ compiler.

4.4 Standard ML

ML was originally designed in the mid-1970s as the meta-language of the Edinburgh LCF theorem-proving system. In 1983 work began on an extensive revision of the language design that led to Standard ML (SML) [Milner, 1984, MacQueen, 1984, Milner *et al.*, 1990]. Various other dialects exist, such as Lazy ML [Johnsson, 1987, Augustsson, 1987] and CAML based on the Categorical Abstract Machine [Cousineau *et al.*, 1985]. ML is essentially a functional language but also includes imperative features.

SML provides structures (i.e. modules) and functors (i.e. functions over structures) for expressing generic programs (Gen ++). Separate signatures express the interfaces of structures (Spe ++). The arguments of functors are constrained by signatures with the possibility of expressing the sharing of structures and types (Par ++). Standard ML does not allow formal specifications of module components but work in this direction has been done in the program development framework EML [Sannella and Tarlecki, 1991, Kahrs *et al.*, 1993, Kahrs *et al.*, 1994] (For 0).

In signatures, no default implementations of exported items may be given (Def −). Structure interfaces are fairly abstract but distinguish between values and parameter-less functions (Abs +). There is a clear distinction between built-in types (defined in the value language) and user-defined structures (defined in the module language) which makes it difficult to exchange both concepts (Typ −, Con −).

The theory of higher-order functors has been developed but the concept is not yet officially supported (Hig +). The contents of structures may be used by qualification (Qua ++) or imported into the current environment; later definitions override former definitions of a name (Imp ++). Overloading is not at all supported which requires large syntactic overhead by the programmer (Ove −). Signatures may inherit other signatures (Ins ++) and structures may inherit other structures (Ini ++).

By the polymorphic type system of SML, all runtime values must be of the same size, if necessary by representing them as pointers to heap objects. All SML compilers implement functors as run-time functions that take structures (records) as arguments and return them as results. The runtime efficiency and representation efficiency of functor-based programs is therefore low (Run −, Rep −). There is no SML support for restricting the scope of functor names in different libraries (Nam −).

As pointed out in [Appel, 1993], there is a lack of good tools for the management of separately compiled source files (Man 0). Since code is only generated for functor definitions, code sharing of functor instantiations is not much of an issue (Sha ␣). While it is usually possible to call in SML programs functions compiled in other languages, it is not easy to embed SML functions into applications written in other languages (Lin 0). The module language is defined separately of the value language but still contains phrases of the value language (Ope 0).

SML has a complete formal definition by an operational semantics [Milner *et al.*, 1990] (Sem ++). There exists a variety of compilers for SML, but not all implement the full definition and some differ in language details (Ava +).

Among all evaluated languages, SML certainly provides the most powerful module system based on a sound semantics. [Appel, 1993] gives a well-founded critique of SML in the light of 10 years' experience with the language. [Berry, 1993] describes lessons learned from the design of the Edinburgh SML library and shows ways in which the language hinders the construction of large libraries (in particular by the lack of higher order functors). [Shao and W., 1993] and [Appel and MacQueen, 1994] describe optimization techniques for the separate compilation of SML modules by the automatic inference of the minimum import interface and by incremental recompilation, respectively.

4.5 A#

The motivation for the development of A# [Watt *et al.*, 1994] was to provide an improved extension language for the computer algebra system AXIOM (the former Scratchpad II [Scratchpad, 1988, Watt *et al.*, 1988]) developed at IBM research laboratories. A# has been designed to model the rich relationships among mathematical structures emphasizing generality, composability, and efficiency. The A# compiler generates Lisp or C code that is compiled to machine code.

There does not exist a concise definition of A#; our evaluation is based on the user manual [Watt *et al.*, 1994] that gives many examples but is not able to answer all semantic questions. On the one hand, the language claims to be "statically typed", on the other hand "types are first class and may be constructed dynamically and manipulated in the same way as any other values"; in fact functions may return types and types may depend on computable values (e.g. integers modulo n). Provided that all type checking occurs at compile time, the language is probably not in all aspects strongly typed.

A# supports domains (abstract datatypes and packages) and functions over domains ("parameterized domains") for expressing generic programs but as outlined above the language seems not in all aspects strongly typed (Gen +). Categories are separate entities that describe the interfaces of domains (Spe ++). The parameters of domains are constrained by categories without the possibility of specifying type sharing (Par +).

There is no way of formally specifying the behavior of operations in categories (For −). Categories may contain default implementations of operations (Def ++). Categories are abstract but it is not possible to express constants by parameter-less functions (Abs +). All built-in types can be replaced without difference by user-defined abstract datatypes (Typ ++). However, the same does not hold for type constructors where only arrays and lists may be redefined without difference for the user (Con 0).

It is possible to write higher-order functions over domains and thus higher-order parameterized domains (Hig ++). It is not possible to refer to the contents of a domain by qualification (Qua −) i.e. abstract datatypes are "weak" and have to be imported into the current environment (Imp ++). Overloading is fully supported (Ove ++); the inheritance of categories is possible (Ins ++) as well as that of domains (Ini ++).

Parameterized domains are just functions; all runtime values have to be of the same size, runtime and representation efficiency of parameterized domains are therefore low (Run −, Rep −). There is no possibility to constrain the scope of domain names in different libraries (Nam −). Apart of the standard Unix tools, there is no support for the management of source code (Man 0); since no instantiated code is generated, code sharing among different users is not much of an issue (Sha ␣). A# code may call external C functions and C programs may call *A#* functions (Lin ++).

The domain/category language is tightly bound to the programming language (Ope −) of which no formal language definition is given (Sem −). There exists a single compiler for A# which has to be commercially purchased (Ava 0).

Among the evaluated languages, A# has certainly the most flexible type/module system. However, it lacks a precise semantics and is probably not strong enough to capture all typing/interface errors at compile time.

4.6 Haskell

In 1987 a committee designed the lazy functional programming language Haskell [Hudak *et al.*, 1992] as an effort to overcome the diversity that had arisen from the existence of more than a dozen rather similar functional languages. Since then various freely available implementations of Haskell have come into existence.

For programming in the large, Haskell supports non-generic modules similar to Modula-2 [Wirth, 1985] or Ada. While not directly supported by the module language, genericity can be expressed by *type classes* (that are similar to the categories in A#) (Gen +). A class declaration introduces the name of a class and the interfaces of the methods of this class.

Instance declarations associate concrete types to classes by mapping abstract class methods into concrete functions. If the instance declaration of a type with respect to a class is visible in a scope, the compiler automatically replaces applications of the abstract class methods to objects of this type by applications of the concrete functions. However, a type can be declared only in one way as an instance of a particular class (Spe +). Class definitions can be parameterized by types from other classes but without the possibility of specifying type sharing (Par +).

There is no possibility of formally describing the behavior of class methods (For −). Classes may contain default implementations of their methods (Def ++). Classes are fairly abstract but distinguish between values and parameter-less functions (Abs +). All primitive operations are defined as class methods; it is therefore easy to replace primitive types by user-defined types (Typ ++). However, since classes may only be parameterized by types, the same does not hold for type constructors (Con −−).

Higher order classes are not supported (Hig −). In fact, Haskell classes only implement weak abstract datatypes in the sense that class qualification is not possible (Qua −) but all class methods are imported into the current environment (Imp ++). Actually the introduction of classes has been motivated by the desire to provide a structured way of overloading functions in the context of a polymorphic type system [Wadler and Blott, 1989] (Ove ++).

By using context clauses, classes may inherit methods from other classes (Ins ++); instance declarations are structured correspondingly but method definitions may not be overridden (Ini +). Because of the polymorphic type system, in current Haskell implementations all runtime values have to be of the same size. Classes are implemented by function tables at runtime, which yields low runtime efficiency and representation efficiency (Run −, Rep −).

Haskell provides modules as a concept to restrict the namespace of objects but there are no means of restricting the scope of module names (Nam −). Management of source code is only supported by the standard Unix tools (Man 0); since no code is generated for class instantiations, code sharing is not much of an issue (Sha ␣). While some Haskell implementations allow to link external functions to a Haskell program, the reverse does by the lazy semantics of the language usually not hold (Lin 0).

The Haskell class system [Hammond and Blott, 1989, Hall *et al.*, 1994] is tightly bound to the lazy programming language (Ope −) for which a formal semantics exists [Peyton Jones and

Wadler, 1991, Hammond and Hall, 1991] (Sem ++). While the language is not industrially supported, there are several free Haskell compilers available (Ava +).

4.7 Summary

Summarizing, we draw the following conclusions:

1. The SML functor notion is the most powerful genericity concept that is safe and has a clear formal basis. Still for large scale software development it requires several extensions, like overloading, higher-order functors and namespaces.
2. While the SML language definition clearly separates the module language from the core language, this distinction is not carried forward to actual implementations. Using the SML module system also means using the SML core language.
3. Only Ada/C++ implement functor-like concepts without dramatic loss of efficiency by code instantiation at compile time. Still mechanisms are needed to avoid replicated code generation for identical instantiations, not just within a single program, but also across multiple users of a generic library.

While research languages (like SML) provide the theoretical concepts for genericity based on functors, they lack the implementation performance and spread of industrially supported languages (like C++).

Chapter 5

Project Contents

In this chapter, we describe the actual contents of the project i.e. what we plan to do and how the results will meet the proposed goals. The core ideas can be summarized as follows:

1. **Functors.** A functor-based module system is developed that is independent of a core language and may be thus integrated with *any* core language. Dependencies between modules are *only* expressed on the top-level by passing modules as functor arguments; reusability is thus maximized. All types and constructors are defined by the functor language and are open for the programmer; implementation flexibility is thus maximized.
2. **Specifications.** Formal propositions in functor specifications may serve as precise contracts between programmer and user *and* between programmer and compiler. Specification of functor components are declarative; reasoning about them is possible in a high-level form. Implementations of functor components are written in the core language and may be imperative. Programming is therefore possible in the most efficient way without being hampered by a formal program development framework. Anyway the "verification gap" of programs is narrowed to the correspondence between functor specification and functor implementation.
3. **Instantiations.** References to components of functor arguments may be resolved at compile-time; genericity does then not imply runtime overhead. Instantiation code is automatically shared among all users; a code explosion in large-scale application of functor libraries is avoided (in combination with the optional run-time instantiation of functors). Dependencies between functor code and instantiated code in the core language are resolved without interference by the programmer.
4. **Application.** The module system is developed as an *open library* for integration with other languages and for use by other developers; a formal reference semantics will be provided. A basic functor library will be implemented demonstrating the use of the system for application programs.

On the one hand we do not intend to develop a toy system just for "proof of concept". Our objective is a system that is useable for serious programmers in challenging applications. On the other hand we commit ourselves to core goals that may be achieved with a reasonable amount of effort but also lead the way to extensions in continuation projects.

5.1 General Picture

We propose to develop a functor-based module system where programming libraries may be constructed of several packages as follows (see Figure 5.1):

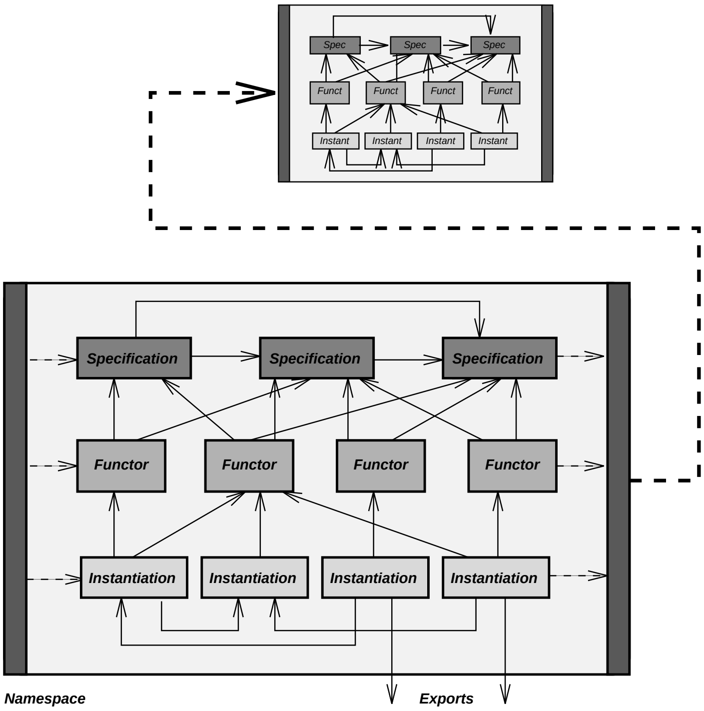

Figure 5.1: A Generic Library

1. **Namespaces.** Every package is encapsulated in a namespace that has a unique name within the library. A namespace may contain as components specifications, functors, instantiations (and other namespaces), each of which is declared in a separate source file.

   Entities may refer to entities in other namespaces only by qualification with the namespace; every package has an export/import interface that explicitly constrains this non-local use of namespace components.

2. **Specifications.** Every specification is part of a denoted namespace and has a unique name among the specifications in this namespace. It describes the entities exported from a module/functor (which may be modules/functors themselves) by listing their syntactic interfaces and formulating their semantic properties.

   The specification can refer to other specifications but *not* to functors or instantiations.

3. **Functors.** Every functor describes how a module is created from argument modules[1]. The functor interface is described by the (names of the) argument specifications and by the (name of the) result specification.

   The functor describes the *implementation* of the entities listed in the result specification. The compiler can easily check the correspondence of the syntactic interfaces between specification and implementation. However, part of the functor are (at least conceptually) also formal *proofs* that the implementation meets the semantic properties of the specification.

   Every functor is part of a denoted namespace and has a unique name among all functors in this namespace with the same interface (the overloading of a name for functors with different interfaces is possible).

[1]Actually the arguments/results may be functors themselves.

The functor may refer to specifications but *not* directly to other functors or instantiations. *All* interaction between functors has to be expressed indirectly by passing arguments to functors.

4. **Instantiations.** Every instantiation describes how a module is created by instantiating functors with argument modules. The instantiation is part of a denoted namespace and has a unique name among all instantiations within this namespace.

   It is only an instantiation that may refer to functors or other instantiations. An instantiation describes how a module is constructed by combining functors; every module is the result of such an instantiation. As a starting point, there exist several predefined modules that represent the interface to the machine and to the operating system, respectively.

   Every instantiation may be *exported* into the environment by creating a library that exports the entities listed in the specification of the result module. For instance, in a C environment a corresponding archive of object modules and a header file are created that may be used in any C program.

In the following subsections, the design of the module system and its implementation are described in more detail.

Of course, this description just reflects a preliminary state of the design; details may and will be changed during the actual development process. Furthermore, the description will be given in ad-hoc pseudo syntax. The actual system will be based on a library operating on abstract syntax trees such that concrete notations can and will be easily changed.

5.2 Namespaces

Every entity (specification, functor, instance) in the module system belongs to a particular namespace. Namespaces are important to integrate different packages without name clashes and to manage different releases of the same package. Namespaces were introduced in the latest draft of the C++ standard [Stroustrup, 1994]; this section mainly presents a reformulation and generalization of these concepts. Also the "post facto extensions" of domains in A# resemble namespaces in certain aspects [Watt *et al.*, 1994].

Namespaces are not first-order objects i.e. may not be components of specification or modules. Before a namespace may be used, it has to be introduced by a namespace declaration. For instance, the declaration

```
namespace Lib = {
  import Lib1
  import Lib2::X
  use Y = Lib2::Y
  use L = Lib3
  export F, S
}
```

introduces a namespace `Lib` that inherits all components of `Lib1`, and the component `X` of `Lib2`. Furthermore the declaration allows its members to refer to `Lib2::X` (by the alias `Y`) and to all components of `Lib3` (by qualification with the namespace alias `L`). External namespaces may be only referenced within such namespace declarations; every package thus may thus easily restrict the use of other packages. Correspondingly, `Lib` exports the names `F` and `S` for use in other namespaces. Namespaces may be also nested like in

```
namespace Lib::N = {
  ...
}
```

where a namespace `N` is declared within `Lib`.

Namespaces are *open* i.e. they are later extended by declarations of namespace components. For instance, the declaration

```
spec Lib::S = {
  ...
}
```

extends `Lib` by the specification `S`. Likewise, the functor declaration

```
functor Lib::F[ ... ]: S = {
  ...
}
```

extends `Lib` by the functor `F`. Members of the same namespace may refer to each other without qualification, thus `F` may use the name `S`, while in other namespaces only `Lib::S` is known.

Every declaration of a namespace member may use a directive to access members of other namespaces without qualification provided that the namespace declaration has allowed access to the namespace by a corresponding `use` clause. For instance,

```
import L::S
functor Lib2::F[ ... ]: S = {
  ...
}
```

declares that `Lib2::F` may refer to `L::S` simply as `S`. Correspondingly, a directive

```
import L
functor Lib2::F[ ... ]: S = {
  ...
}
```

declares that `Lib2::F` may refer to all members of `L` without qualification.

The namespace concept is essentially a mechanism for structuring a set of global declarations that are split across multiple source files. While important for the pragmatic use of a module system in large-scale applications, it does actually not represent an additional dimension for the underlying semantics. In the following sections, we will therefore ignore namespaces and treat all declarations as global.

5.3 Specifications

We will outline the proposed system of specifications by describing how it can be used to model the types and type constructors that are built-in in most programming languages. By this process we will motivate the necessity of various features that will be introduced in the course of the exposition. We will sketch how to model

1. Booleans
2. Numbers
3. Sequences
4. Arrays
5. Products and Sums
6. Functions

The goal of these specifications is to serve as precise contracts between programmers and between programmer and compiler. It is based on similar ideas as Extended ML but is on the one hand humbler but on the other hand more challenging: it is humbler because specifications may serve as interfaces between module implemented in any core language even if they cover not all aspects of the core language. Thus the implementation itself is not constrained by a formal design methodology; nevertheless other programmers respectively the compiler may rely on these contracts. Our goal however is more challenging, because we will not base the development on the built-in types of a fixed core language but model the complete type system from scratch and thus open it to the programmer.

The following description of the specification language is concrete but informal; part of the project will be the detailed design and the construction of a formal semantics as the basis of the implementation.

5.3.1 Booleans

In this subsection, we describe how the datatype "Boolean" may be specified in a program such that any user-defined type that meets the specification can serve as a full substitute for the corresponding built-in type. Due to the fundamental role of this type, a correct specification is already challenging.

The view taken by most core languages is that the type Boolean is special in the sense that for a Boolean value v the command

`if` v `then` c_1 `and` c_2

and/or for values v_1, v_2 of any type T the operator

`if (`v`,` v_1`,` v_2`)`

is available. Our first goal is to overcome the special role of this type in the context of *conditional constructs* and derived constructs (such as while loops). We therefore need the possibility to convert an abstract datatype Boolean into the corresponding built-in type in order to compute *decisions* by selecting one of two program branches.

Second, we want to express *algorithmic knowledge* by formal sentences in first-order predicate logic. However, these formal sentences shall also have a computational nature such that they may also serve as run-time *assertions* and as *default implementations* of functions. Our logic will therefore be partially based on predicates that are Boolean-valued program functions. The specification of Boolean values must therefore be itself based upon the datatype Boolean. We show how to overcome this apparently vicious circle.

In this context, another fundamental difficulty is that even the concept of *equality* is not a priory clear. We have no possibility to decide whether two objects of an abstract datatype are equal or not except by a user-defined function that is supplied as part of this datatype (only this function is able to look into the internal structure of the objects). In other words, for the compiler and any user a datatype is fully abstract i.e. nothing but a *name*: of values of this type nothing is known, nothing can be decided and nothing can be extracted apart from the functions provided by the datatype.

With these problems in mind, we are now going to sketch the specification of the abstract datatype of Boolean values. We begin with the module specification

```
spec Type = {
  type Type
}
```

that introduces a type name `Type`. Since no operation is defined and no knowledge is available on this type, this specification is satisfied by any module that implements any type of this name. The rationale for this specification is to provide a basis for functors that may operate on arbitrary arguments.

The module specification

```
spec BoolCore = {

  type Bool
  type Type = Bool

  val true: Bool
  val false: Bool

  proc if(Bool, proc(), proc())
  prop condproc(Bool, if, true, false)

}
```

introduces the abstract datatype `Bool` and defines for this type a synonym `Type` such that any module that satisfies `BoolCore` also satisfies the specification `Type`. Actually the definition `type Type = Bool` is translated into

```
type Type
prop Bool = Type
```

A proposition of the form

`prop` $type_1$ `=` $type_2$

introduces a sharing specification that asserts that two names refer to the same underlying type.

The specification also introduces two names `true` and `false` for values of type `Bool`. This does *not* yet imply that the values are different (both names might stand for the same object) or that these are the only `Bool` values.

`BoolCore` defines a procedure (command) `if` that takes as arguments a `Bool` value and two parameter-less procedures. The predefined proposition

`prop condproc(`B`,` p`,` t`,` f`)`

asserts that the execution of the procedure $p(t, c_t, c_f)$ is equivalent to the execution of the command c_t and that the execution of $p(f, c_t, c_f)$ is equivalent to the execution of c_f i.e. that p fulfills the usual semantics of the conditional command.

Every functor with a parameter `C:BoolCore` may then declare

```
import C
```

to import the contents of the `BoolCore` module into the current environment. Consequently he is entitled to use a statement like

`if` p `then` c_1 `else` c_2

that just represents syntactic sugar for the call of the imported procedure. A pre-defined instantiation implements this functionality with the facilities of the core language (i.e. `Bool` values are the booleans of the core language and the procedure `if` is translated into the corresponding core language construct). Based on this module, the programmer may define new implementations with the same functionality and the same use. For instance, the functor

```
functor RevBoolCore[B:BoolCore]: BoolCore = {
  type Bool = B.Bool
  type true = B.false
  type false = B.true
  fun if(x:Bool, y:Bool, z:Bool) = B.if(x, z, y)
}
```

provides a replacement where the implementations of `true` and `false` are exchanged; any `BoolCore` argument `B` of any functor may now be replaced by `RevBoolCore[B]` without difference in use.

The functor specification

```
spec If[C: BoolCore, T:Type] = {

  fun if(C.Bool, T.Type, T.Type): T.Type
  prop condfun(C.Bool, T.Type, if, C.true, C.false)

}
```

allows to extend the functionality of Booleans by a generic `if` operator that returns either of two values of a denoted type. Arguments to a functor `F:If` are the core definition of Booleans and the type of the returned value. The predefined proposition `condfun` asserts that the denoted procedure fulfills the laws of the conditional operator.

The above specification `If` is higher-order i.e. it does not describe the interface of a module but that of a functor. A functor `F:If` could be essentially also described by the interface `F[BoolCore,Type]:If2` with the first-order specification

```
spec If2 = {

  module C: BoolCore
  module T: Type

  fun if(C.Bool, T.Type, T.Type): T.Type
  prop condfun(C.Bool, T.Type, if, C.true, C.false)

}
```

However, the `If` specification is stronger in that, in addition to describing the interfaces of the argument modules and of the result module, it also demands sharing among arguments and result. For any functor `F:If2` and application `I = F[C,T]`, we then know

```
prop I.C = C
prop I.T = T
```

which is not demanded by the corresponding first-order specification. We may base, however, the specification `If` on `If2` by the clause

```
spec If[X: BoolCore, Y:Type] = If2[C = X, T = Y]
```

where the right-hand side is derived from `If2` by introducing the additional sharing properties given above.

Based on these specifications, we may define the datatype Boolean as follows:

```
spec Bool = {

  module Core: BoolCore
  import Core

  functor If: If
```

```
  functor If = If[Core,_]
  import If[Core]

  fun id(x:Bool): Bool = x
  fun not(x:Bool): Bool = if(x, false, true)
  fun and(x:Bool, y:Bool): Bool = if(x, y, false)
  fun or(x:Bool, y:Bool): Bool = if(x, true, y)
  fun implies(x:Bool, y:Bool): Bool = if(x, y, true)
  fun equiv(x:Bool, y:Bool): Bool = if(x, y, not(y))

  prop equality(Bool, equal)
  prop constructors(Bool, (id, true), (not, false))

}
```

This specification consists of the following components:

1. **Modules.** The core definition of boolean values is included as a submodule `Core` and imported into the current environment. Any `Bool` module therefore also fulfills the specification `BoolCore`.
2. **Functors.** The module incorporates an `If` functor; any user of a `Bool` module `B` may therefore declare

   `import B.If[`T`]`

   that introduces the `if` operator with return values of type T`.Type`. The module itself introduces by `import If[Core]` the `if` operator on `Bool` values.
3. **Default Implementations.** The specification

   f`(`x`:`S`,...):`T` = `e

   is defined as the interface specification

   `fun `f`(`x`:`S`,...):`T

   and the proposition

   `prop forall(`x`:`S`,...)(`eq`(`f`(`x`),`e`))`

   eq is a function declared in a proposition `prop equality(`T`,`eq`)` (see below). If e is a computable expression, it may be used as a *default implementation* of f.

   In the `Bool` case, we circumvent the specification of the behavior of the `Bool` operations by reduction to the conditional operator whose semantics in turn is defined by the `condfun` proposition.
4. **User-Defined Proposition.** The proposition

   `prop `$p(a,\ldots)$

   (where p is a user-defined function with result type T) says $p(a,\ldots)$ *holds* i.e. that the execution of the procedure

   if`(`$p(a)$`, SKIP, ABORT)`

   (where `SKIP` is a predefined null operation and `ABORT` terminates the program) at any time does not change the behavior of the program. The procedure if is the proposition declared in the corresponding `condproc(`T`, `if`, `t`, `f`)` proposition.

   We thus reduce the validity of a proposition to a computational decision that depends on the result of a computer function. However, the arguments of p need not be fully constructive as shown by following paragraphs.
5. **Universal Quantification.** The expression

`forall(`x`:`T`,...)` e

(with $e : S$) is defined as the expression

if`(`$e[x_1/x, \ldots]$`,` if`(`$e[x_2/x, \ldots]$`, ...,` f`),` f`)`

for all values $x_i : T, \ldots$. The function if and the constants t and f are those declared in the corresponding `iffun` proposition.

This expression may be computable at compile-time (if all values of $T, \ldots$ are known), at run-time (if $T, \ldots$ has only a finite number of values that can be enumerated), or not all (if $T, \ldots$ has an infinite number of values).

6. **Existential Quantification.** The expression

   `exists(`x`:`T`,...)` e

   is defined as the expression

   if`(`$e[x_1/x, \ldots]$`,` t`, if(`$e[x_2/x, \ldots]$`,` t`, ...))`

   for all values $x_i : T, \ldots$. The function if and the constants t and f are those declared in the corresponding `iffun` proposition.

   The same remarks concerning computability hold as for universal quantification.

7. **Implicit Definition.** The expression

   `some(`x`:`T`)` e

   is defined as any value

   $y : T$

   such that the proposition $e[y/x]$ holds where `exists(`x`:`T`,...)` e is assumed.

8. **Equality.** The proposition

   `prop equality(`T`,`eq`)`

   says that the function eq represents *equality* over T i.e.

   (a) *Reflexivity.* For every function $f(S_1, \ldots) : T$ and all arguments $x_1 : S_1, \ldots$, $eq(f(x_1, \ldots), f(x_1, \ldots))$ holds.

   (b) *Symmetry.* For all $x : T$, $y : T$, $eq(x,y)$ implies $eq(y,x)$,

   (c) *Transitivity.* For all $x : T$, $y : T$, $z : T$, $eq(x,y)$ and $eq(y,z)$ implies $eq(x,z)$,

   (d) *Equivalence.* For every $x : T$ and $y : T$ with $eq(x,y)$, every application of x can be replaced by an application of y without changing the behavior of the program.

   Our notion of reflexivity generalizes the mathematical notion because f is a program function that may return for the same arguments different results; sometimes these results are nevertheless considered equal. Take e.g. the list expression `cons(h,t)` that returns on each application a new list cell that could be distinguished by its unique memory address. However, any observer that is only given access to the operators `isnil`, `head` and `tail` must consider the cells as equal.

9. **Constructors.** The proposition

   `prop constructors(`T`, (`p_1`,` c_1`,` d_{11}`, ...), ...)`

   says that for every value $v : T$ exactly one of the sentences

   if`(`p_1`(`v`),` eq`(`v`,` c_1`(`d_{11}`(v), ...)),` f`)`
   ...

holds where *if* and f are those values declared in the corresponding `condfun` proposition.

Every $c_i(S_{i1}, \ldots) : T$ is then called a *constructor* of T; p_i is the corresponding *predicate* and the $d_{ij}(T) : S_{ij}$ are the corresponding *selectors*. Nullary constructors are *constants* of T.

If the constructors of a type are given, pattern matching (i.e. a finite number of case distinctions) may be applied to decompose all of its values. Functions over this type may be defined and proofs may be constructed by applying *induction* over the structure of the type. Likewise, equality of such a type is defined provided that equality is defined on those types on which its constructors depend. If a type has only constant constructors, its values may be even enumerated at compile time.

We omit the details how the compiler may map the functions and operators defined on Booleans into the concrete syntax of the core language. Since this mapping is rather straightforward, we will from now on make free use of syntactic sugar in particular write `if` ...`then` ...`else` ... for an application of the `if` operator and `==` for the `equal` predicate.

5.3.2 Numbers

We are now going to elaborate the specification of natural and integer numbers starting with the core definition of natural numbers:

```
spec NatCore = {

  module Bool: Bool
  type Bool = Bool.Type

  type Nat
  type Type = Nat

  val zero: Nat
  fun succ(Nat): Nat
  fun pred(Nat): Nat
  fun iszero(Nat): Bool
  fun notzero(Nat): Bool = Bool.not(iszero(x))
  fun equal(Nat, Nat): Bool

  prop constructors(Nat, (iszero, zero), (notzero, succ, pred))
  prop equality(Nat, equal)

}
```

The specification incorporates a submodule of specification `Bool` on which the exported predicates are based. Natural numbers are defined inductively using the constructors `zero` and `pred` and the corresponding selectors and test predicates. Based on this knowledge and its declaration as the equality predicate, the function `equal` could be even automatically constructed.

Using the core definition, the common functions and predicates on natural numbers may be defined as follows:

```
spec Nat = {

  module NatCore: NatCore
  import NatCore.Bool
  import If[NatCore]
```

```
    type Nat = NatCore.Type
    type Type = Nat

    val one: Nat = succ(zero)
    val two: Nat = succ(one)
    ...
    val sixteen: Nat = succ(fifteen)

    fun less(x:Nat, y:Nat): Bool =
      (iszero(x) and notzero(y)) or less(pred(x), pred(y))

    fun lesseq(x:Int, y:Int): Bool =
      equal(x,y) or less(x,y)

    fun sum(x:Nat, y:Nat): Nat =
      if iszero(x) then y else sum(pred(x), succ(y))

    fun diff(x:Nat, y:Nat): Nat =
      if iszero(y)
        then x
        else diff(pred(x), pred(y))
    prop input(diff(x:Nat, y:Nat), not(less(x,y)))

    fun minus(x:Nat) = diff(zero,x)
    ...
  }
```

The specification includes `NatCore` as a submodule and imports its definition of Boolean values for convenience. Likewise the conditional operator is defined for natural number results by importing `If[NatCore]`. The specification of all functions are constructive and may therefore be used as default implementations.

The function `diff` uses an *input condition*

`prop input(`f`(`x_1`:`T_1`, ...), `c`(`x_1`, ...))`

i.e. the result of any application of f is only defined if the condition c holds on the function arguments. Besides serving as user documentation and as the basis of formal reasoning, the input condition may be used to automatically construct *runtime assertions* that precede any application of the function (provided that the definition of the predicate is constructive).

As for Booleans, we will from now on use common syntax for the defined numeric operations i.e. `<` for `less`, `+` for sum, etc. In Subsection 5.3.3 we will deal in more detail with the question how *numeric literals* may be handled.

We now turn to the specification of the core of integer numbers:

```
  spec IntCore = {

    module Bool: Bool
    import Bool

    type Int
    type Type = Int

    val zero: Int
    fun succ(Int): Int
    fun pred(Int): Int
    fun iszero(Int): Bool
    fun notzero(x:Int): Bool = not(iszero(x))
    fun equal(Int, Int): Bool
```

```
  prop vars(x: Int)

  prop equality(Int, equal)
  prop equal(x, succ(pred(x)))
  prop equal(x, pred(succ(x)))
  prop not(equal(succ(x),x))
  prop not(equal(pred(x),x))
  prop not(equal(pred(x),succ(x)))
  prop iszero(x) == equal(x, zero)

}
```

The main difference to the specification of natural numbers is that integers cannot be described by a simple term algebra of constructors. Instead we have to provide axioms that uniquely characterize the basic integer operations. Based on the definition of **equal** as the equality predicate, we thus describe the relationships between x, **succ**(x), **pred**(x). The declaration

prop vars(x:T,...)

introduces universally quantified variables and denotes their types. Any proposition **prop** p has consequently to be read as

prop forall(x:T,...) p

Using the core specification, we may define integers as follows

```
spec Int = {

  module IntCore: IntCore
  module Bool: Bool = IntCore.Bool

  import IntCore
  import BoolCore
  import If[IntCore]

  fun gtzero(x:Int): Bool
  fun ltzero(x:Int): Bool

  prop gtzero(succ(zero))
  prop not(gtzero(zero))
  prop ltzero(pred(zero))
  prop not(ltzero(zero))
  prop not(iszero(x)) implies (gtzero(x) == gtzero(pred(x)))
  prop not(iszero(x)) implies (ltzero(x) == ltzero(succ(x)))

  fun sum(x:Int, y:Int): Int
  prop sum(zero, y) == y
  prop sum(succ(x), y) == sum(x, succ(y))
  prop sum(pred(x), y) == sum(x, pred(y))

  fun diff(x:Int, y:Int): Int
  prop sum(diff(x,y),y) == x

  fun less(x:Int, y:Int): Bool = ltzero(diff(x, y))
  fun lesseq(x:Int, y:Int): Bool = equal(x,y) or less(x,y)
  ...

}
```

This specification extends `IntCore` by importing a corresponding submodule. It also imports the associated definition of Boolean values and introduces the conditional operator with integer result values.

We specify additional predicates `gtzero` and `ltzero` by axioms. While these axioms are sufficient for reasoning and they also might be in principle used to construct executable definitions, the resulting functions would be hopelessly inefficient. Any implementation of `Int` will therefore better provide its own definitions of these functions. The same is true for the axiomatic characterization of `sum` and `diff` (although the axioms resemble pattern-matching programs, they do not directly yield executable definitions).

5.3.3 Sequences

The specification of *sequences* of values of the same type is rather straight-forward:

```
spec Sequence = {

  module Elem: Type
  module Bool: Bool

  type Sequence
  type Type = Sequence
  type Elem = Elem.Type
  type Bool = Bool.Type

  val nil: Sequence
  fun cons(Elem, Sequence): Sequence
  fun isnil(Sequence): Bool
  fun notnil(Sequence): Bool = Bool.not(isnil)
  fun first(Sequence): Elem
  fun rest(Sequence): Sequence
  fun equal(Sequence, Sequence): Bool

  prop constructors(Sequence, (isnil,nil), (notnil, cons, first, rest))
  prop equality(Sequence, equal)

}
```

This specification uses the submodule `Elem` to denote the type of element values and a submodule `Bool` to denote Boolean values. `Elem` is of specification `Type` i.e. no knowledge is assumed on this type within this specification. However, any functor constructing a module of this specification needs equality on the element type to define equality on the sequence type, for instance

```
functor List[E:EqType]: Sequence = {
  module Elem: Type = E
  ...
  fun equal(x:Sequence, y:Sequence) =
    ... E.equal(first(x),first(y)) ...
}
```

A first application of sequences is the introduction of numeric literals like '123' in the context of abstract datatypes. We proceed by extending the specification `Nat` with sequences of natural numbers:

```
spec Natural = {

  module Nat: Nat
  module Bool: Bool = Nat.Bool
```

```
  module Seq: Sequence
  prop Seq.Elem = Nat
  prop Seq.Bool = Bool

  import Nat
  import Bool
  type Seq = Seq.Type

  fun seqnat(b:Nat, d:Nat, s:Seq): Nat =
    import Seq in
    if isnil(s) then d else d+b*seqnat(b, first(s), rest(s))
  prop input(seqnat(b:Nat, d:Nat, s:Sequence), d < b)

  fun litnat(b:Nat, s:Seq) = seqnat(b, Seq.first(s), Seq.rest(s))
  prop input(litnat(b:Nat, s:Sequence), zero < b and not(Seq.isnil(s)))

  pragma literal(Nat, litnat, cons, nil, (zero, one, ..., fifteen))

}
```

This specification incorporates two submodules `Nat` and `Seq` both of which contain their own `Bool` modules. In order to combine the results of `Nat` predicates and the results of `Seq` predicates (as in the `input` condition of `litnat`) we have to demand that `Nat` and `Seq` *share* the same `Bool` module. This is described by a specification of the form

`prop` M_1 `=` M_2

The function `litnat` describes how a sequence of `Nat` values is interpreted as a numeric literal with respect to a certain base b. This function uses an auxiliary function `seqnat` which imports `Seq` by a construct

`import` M `in` e

restricts the effect of the import to the expression e.

The pragma

`pragma literal(`T`,` f`,` c`,` n`, (`c_0`, ...))`

tells the compiler that within a module that imports the contents of a `Natural` module a literal string 102 is interpreted as the T value

`litnat(`c_{10}`, cons(`c_1`, cons(`c_0`, cons(`c_2`,` n`))))`

i.e. after resolving the definition of `litnat` as

$$c_2 + c_{10} * (c_0 + c_{10} * c_1)$$

This concept can be extended to integer numbers as follows:

```
spec Integer = {

  module Int: Int
  module Bool: Bool = Int.Bool

  module Nat: Natural
  module Seq: Sequence = Nat.Sequence

  import Int
  import Bool
```

```
  type Nat = Nat.Type
  type Seq = Seq.Type

  fun natint(n:Nat): Int =
    import Nat in
    if iszero(n)
      then Int.zero
      else succ(natint(pred(n)))

  fun intnat(i:Int): Nat
  prop input(intnat(i:Int), i >= zero)
  prop forall(n:Nat) Nat.equal(intnat(natint(n)), n)

  fun litint(s:Seq): Int = natint(Nat.litnat(s))
  pragma literal(Int, litint, Seq.cons, Seq.nil,
                 (Nat.zero, Nat.one, ..., Nat.fifteen))

}
```

This specification contains a `Natural` module and describes by two functions `natint` and `intnat` an isomorphism between natural numbers and the positive integer numbers. If a module imports an `Integer` module, the literal -102 is now interpreted as

- `natint`$(c_2 + c_{10} * (c_0 + c_{10} * c_1))$

where the c_i are `Nat` values, the operations $+$ and $*$ denote `Nat.sum` and `Nat.prod`, and $-$ denotes `Int.minus`.

5.3.4 Arrays

An *array type* is characterized by direct access to any of its elements through a numeric index, as well for reading as for writing. It may axiomatized in a specification as follows:

```
spec ArrayCore = {

  type Array
  type Type = Array

  module Nat: Natural
  module Elem: EqType
  module Bool: Bool

  import Nat
  type Elem = Elem.Type
  type Bool = Bool.Type

  fun init(Nat, Elem): Array
  fun put(Array, Nat, Elem): Array
  fun size(Array): Nat
  fun get(Array, Nat): Elem
  fun equal(Array, Array): Bool

  prop input(get(a:Array, n:Nat), n < size(a))
  prop input(put(a:Array, n:Nat), n < size(a))

  prop vars(n:Nat, e:Elem, a:Array, b: Array)

  prop size(init(n, e)) = n
  prop size(put(a, n, e)) = size(a)
```

```
    prop get(init(n, e), m) = e
    prop get(put(a, n, e), n) = e
    prop n <> m implies get(put(a, n, e), m) = get(a, m)

    prop equal(a, b) == (size(a) == size(b) and
         forall(n:Nat) n < size(a) implies get(a,n) == get(b, n))
    prop equality(Array, equal)

  }
```

While this specification captures the mathematical interpretation of arrays, it fails to cover the essence of their practical use. In real applications, arrays are not constructed by a function that takes an array and an element and returns a result array that differs from the argument at one index position but by a procedure that destructively updates the input.

We may capture this behavior by the following definition of a `set` procedure:

```
  spec Array = {

    module Core: ArrayCore
    import Core

    proc set(inout Array, Nat, Elem)

    prop input(set(a:Array, n:Nat, e:Elem), n < size(a))
    prop output(set(a:Array, n:Nat, e:Elem),
      size(a) = size(old(a)) and
      get(a, n) = e and
      forall(m:Nat) n <> m implies get(a, m) = get(old(a), m))

  }
```

A core language operation `a[i] = e` is then interpreted as syntactic sugar for the application `set(a, i, e)`.

The first parameter of `set` is marked as `inout`; it is a *transient argument* i.e. a variable that has after the procedure call another value than before. The corresponding *output condition* c declared in

`prop output(`p`(`x`:`T`, ...), c)`

describes the condition that these variables meet after the procedure has returned. This condition may refer to the original value of a transient argument v as `old(`v`)`.

Still the use of destructive operations raises problems because of *pointer sharing*. Complex objects are commonly represented as pointers to memory regions (that may themselves contain pointers to other regions). In this way only small pointers have to be handled instead of unwieldy data structures. In a definition like `v1 = v2` then only pointers have to be duplicated instead of the whole data structure. However, this simplicity is traded by the disadvantage that there is no more one to one correspondence between variables and objects. Any destructive operation on one variable (say `v1`) may also affect the values of other variables (say `v2`).

Provided that an `Array` is represented by a pointer, an application of `set` is therefore only valid if its first argument is the *only* reference to the underlying memory structure. There are two approaches to ensure this input condition:

1. **Reference Counting.** The underlying data structure counts the number of references to it. This purpose may be achieved by overloading the *definition (assignment) operator* `=`

```
proc assign(inout Array, Array)
prop assignment(Array, assign)
```

where the proposition

`prop assignment(`T`, `p`)`

denotes that any definition v `=` e of type T has to be interpreted as the application of the procedure p`(`v`, `e`)` and that this procedure has the output condition

`prop output(`p`(v:`T`,e:`T`), v == e)`

where equality must be defined on T.

2. **Linear Types.** The theory of linear types has been developed in [Girard, 1987, Wadler, 1991, Baker, 1995]. The core idea is that the compiler ensures that every variable is used exactly once such that a one-to-one correspondence between variables and objects is maintained. While the universal use of linear types leds to a rather peculiar programming style, the idea may be simply used in a restricted context. A specification

   ```
   fun init(Nat, Elem): single Array
   ```

   declares that the result of `init` is an unshared array that is referenced only by the return value of the function.

   The compiler may check that the `inout` arguments of procedures are not referenced more than once by checking whether they were created by a function with a `single` result and the return value was not used elsewhere except as procedure arguments marked as `inout single`. For instance, the specification

   ```
   proc set(inout single Array, Nat, Elem)
   ```

   ensures that the first argument not only enters but also leaves the procedure without sharing. If the argument is not a priory known to be unshared, a function

   ```
   fun dup(Array): single Array
   prop forall(a:Array) dup(a) == a
   ```

   has to be applied before that returns an unshared copy of its input.

Both approaches have their justification and may be even combined: while the *specification* of `Array` may be based on linear types, its *implementation* may be also based on reference counting. The function `dup` may thus check the reference count of its argument to return directly its input (if only referenced once) or a duplicate of it. Still the implementation may be replaced by a non reference-counting one without changing programs relying on the specification.

Summarizing, we therefore want to support this restricted form of linear types together with the possibility of overloading the definition operator `=`.

5.3.5 Products and Sums

A *product (tuple) type* is defined by a functor that takes as arguments an arbitrary number of types (for the product components) and returns a type constructor and the corresponding component selectors. This behavior is described by the following functor specification:

```
spec Product[ L:<< (<I>, M:Type) >> ] = {

  type Product
  type Type = Product
```

```
    fun product( L(E):<< E.M.Type >> ): Product
    L(E):<< fun component<E.I>(Product): E.M.Type >>

    module Bool: Bool
    fun true(x:Product):Bool.Bool = Bool.true
    prop constructors(Product, (true, product, L(E):<< component<E.I> >> ))

  }
```

A functor `P` described by this specification takes as arguments a list `L` of pairs (I, M) where each I is an identifier and each M is a module of specification `Type`. It returns a module that exports a `Product` type, a constructor function `product` and for every functor argument a selector function `component`I.

The construct

```
L(E):<< ... E.M ... >>
```

repeats the construct enclosed by `<< ... >>` once for every member `E` of `L` where `E.M` denotes the corresponding component of the argument list.

The result module of an application `P((B, Bool), (I, Integer))` therefore fulfills the specification

```
  spec {

    type Product
    type Type = Product

    fun product(Bool.Type, Integer.Type): Product
    fun componentB(Product): Bool.Type
    fun componentI(Product): Integer.Type

    module Bool: Bool
    fun true(x:Product):Bool.Bool = Bool.true
    prop constructors(Product, (true, product, componentI, componentB))

  }
```

Type constructors with a variable number of arguments thus requires a generalization of the concept of functors; while conceptually not very difficult, this flexibility is not provided by the functor languages of ML or A#.

The usual syntactic sugar for record/tuple types may be applied to this specification of Product types. A program expression `(b,i)` thus denotes the function application `product(b,i)` and a selector expression `t.I` actually means `componentI(t)`.

By a similar strategy, the concept of a *sum (disjoint union) type* may be modelled:

```
  spec SumCore[ L:<< (<I>, M:Type) >> ] = {

    type Sum
    type Type = Sum

    L(E):<< fun sum<E.I>(E.M.Type): Sum >>
    proc case(Sum, L(E):<< proc(E.M.Type) >> )
    prop caseproc(Sum, case, L(E):<< sum<E.I> >> )

  }
```

The result of a functor of specification `SumCore` is, for every argument pair (I, M), a constructor function `sumI` that maps a value of type M`.Type` into type `Sum`. The procedure `case` takes a `Sum` value and a list of procedures and selects the procedure to be executed for every possible case that the sum value was constructed. The proposition

`prop caseproc(`T`,` $case$`,` c_1`, ...)`

declares that the execution of $case(c_i(x),\ p_1,\ \ldots)$ is equivalent to the execution of $p_i(x)$.

A case command in the core language like

```
case v of
  isB(x) => p(x)
  isI(x) => q(x)
```

(where `v` is of a sum type with constructors `sumB` and `sumI`) is therefore syntactic sugar for the application `case(v, p, q)`.

Based on the core specification of sum types a *case operator* may be defined with arbitrary return types:

```
spec Case[S:SumCore, R:EqType] = {

  fun case(S.Type, S.L:<< fun(S.M.Type):R.Type >> ): R.Type
  prop casefun(S.Sum, case, S.L:<< sum<S.I> >> )

}
```

The proposition

`prop casefun(`T`,` $case$`,` c_1`, ...)`

declares that the proposition

$$case(c_i(x),\ f_1,\ \ldots) = f_i(x)$$

holds where an equality predicate $=$ must be defined on the result type of the f_i.

The declaration of case procedures/functions with `prop caseproc/casefun` is equivalent ot the declaration of constructor/selector functions with `prop constructors` i.e. each concept can be automatically derived from the other one. However, based on the actual implementation either of them may be more efficient; it is therefore always a good idea to provide both such that the user may select that one that seems more appropriate for his purpose.

Using these functors, we may then construct the sum functor

```
spec Sum[ L:<< (<I>, M:Type) >> ] = {

  functor SumCore: SumCore
  module SumCore = SumCore[ L(E):<< (<E.I>, E.M) >> ]
  import SumCore

  functor Case: Case
  functor Case = Case[SumCore, _]

}
```

that extends `SumCore` by a functor that constructs a `case` operator for every return type.

A special case of disjoint union functors are *enumeration types* consisting of a fixed set of constants. They can be modelled based on the functor specification

```
spec EnumCore[ L:<< <I> >> ] = {

  type Enum
  type Type = Enum

  L(E):<< fun enum<E.I>(): Enum >>
  proc case(Enum, L(E):<< proc() >> )
  prop enumproc(Enum, case, L(E):<< enum<E.I> >> )

}
```

that takes a list of identifiers and constructs a set of parameter-less constructor functions and a corresponding `case` procedure.

5.3.6 Functions

Even the modeling of *function spaces* is not beyond the power of the specification language. A function space is defined by two constructions *abstraction* (that builds a function from a syntactic variable and a syntactic expression) and *application* (that applies the abstraction to a value and returns a result).

However, in its full generality this approach requires the modeling of expression construction and of type checking, tasks that we do not want to perform at runtime but always leave to the compiler. Actually, we are less interested in modeling the abstraction of functions but in the possibility of treating functions as first order objects.

For this purpose we assume that every non-first occurrence of a function name f (i.e. as a function argument or result) is syntactic sugar for the expression `lift(`f`)` and that every application $f(x)$ of a function f passed as a function argument or returned as a function result is syntactic sugar for the expression `apply(`f`, `x`)`. This replacement can be done automatically by the compiler.

Furthermore we assume a specification

```
spec FunctionCore = {

  module Domain: EqType
  module Target: EqType

  type Function
  type Type = Function

  type Domain = Domain.Type
  type Target = Target.Type

  fun apply(Function, Domain): Target
  property function(Function, Domain, Target, apply)

}
```

and a corresponding functor

```
functor FUNCTION[D: EqType, T: EqType]:
  FunctionCore[Domain = D, Target = T]
```

such that the compiler automatically imports `FUNCTION` into every user-defined functor. For every domain type D and target type T, the higher-order occurrence of a function $f(D) : T$ has type f`:FUNCTION[`D`,`T`].Type`. (We only present the one-argument case; by the techniques presented in the previous subsection, this approach can be also generalized to functions with arbitrary numbers of arguments).

The programmer may now use a specification

```
spec Function = {

  module Core: FunctionCore

  type Function
  type Type = Function

  type Domain = Core.Domain
  type Target = Core.Target

  fun lift(f: Core.Function): Function
  fun apply(Function, Domain): Target

  prop vars(f:Core.Function, x:Domain)
  prop apply(lift(f), x) = Core.apply(f,x)

}
```

to build a functor

```
functor FunctionFunct[C: FunctionCore]: Function[Core = C]
```

and import the module `FunctionFunct[FUNCTION[`D`,`T`]]` into the current environment. By providing definitions for `apply` and `lift`, he may thus create his own function type.

This way of modelling function types breaks somehow our original idea of not having any built-in types (`FUNCTION` is actually such a built-in type constructor). This is a consequence of that we consider for convenience the construct *function abstraction* as built-in and not accessible by the programmer. However, this is by no means a principal restriction; function spaces can be also modelled in their full generality (by defining the abstract datatype program expression with associated typing and evaluation functions).

5.3.7 Summary

Our specification-language is based in spirit on the specification language of SML with the following differences and extensions:

1. **Declarations.** Every specification has a name and is declared on the top-level (no nesting of specifications). Specifications may only refer to other specifications but not to functors or modules declared outside the specification.
2. **Higher-Order.** A specification may have (module or functor) parameters; in this case the specification describes a functor interface. These arguments are implicit components of the result module and thus allow to express sharing between functor arguments and module components.
3. **Variable Arguments.** A higher-order specification (and the corresponding functor) may have a variable number of arguments and consequently a variable number of exported components.
4. **Components.** A specification may introduce the name of an abstract datatype or give a synonym for another datatype. It may introduce the interfaces of exported procedures, functions, and values. Transient procedure parameters are explicitly denoted; function result may be declared as singly referenced for use as transient procedure arguments.
5. **Properties.** The specification may declare the properties of exported components; the compiler may rely on these properties for program generation, transformation

and optimization to generate runtime assertions and, in some cases, even default implementations. These properties include:

- `prop` p (the first-order logic proposition p always holds).
- `prop condproc(`B`,` p`,` t`,` f`)` (p is a conditional command).
- `prop condfun(`B`,` f`,` t`,` f`)` (f is a conditional operator).
- `prop equality(`T`,`eq`)` (eq denotes equality on T).
- `prop constructors(`T`, (`p_1`,` c_1`,` d_{11}`, ...), ...)` (T is a free term algebra of the denoted constructors).
- `prop input(`f`(`x_1`:`T_1`, ...),` c`(`x_1`, ...))` (c is the input condition of function/procedure f).
- `prop output(`p`(`x`:`T`, ...), c)` (c is the output condition of procedure p).
- `prop` T_1 `=` T_2 (T_1 and T_2 refer to the same type within a module).
- `prop` M_1 `=` M_2 (M_1 and M_2 refer to the same module).
- `prop assignment(`T`,` p`)` (p is the assignment command on T).
- `prop caseproc(`T`,` c`,` c_1`, ...)` (c is a case command).
- `prop casefun(`T`,` c`,` c_1`, ...)` (c is a case operator).

6. **Pragmas.** The specification may contain compiler pragmas that denote that wherever the specified module is imported, syntactic categories must be treated in a special way. For instance,

   - `pragma literal(`T`,` f`,` c`,` n`, (`c_0`, ...))` specifies how numeric literals are translated into constants of type T.

5.4 Functors

In this subsection, we sketch how functors may implement specifications. Conceptually a functor consists of the following items:

1. **Parameters.** The formal parameters denote modules (respectively functors) on which the current functor depends. The functor interface describes the specifications and the sharing expected from these modules and the specification of the functor result.

   All interaction between modules is expressed by functor arguments. There is no direct dependence between modules.

2. **Imports.** The functor may import the entities listed in the specification of parameter modules into the current module. Imported entities may be used as if they were defined within the current functor or as if they were bult-in into the core language, respectively.

3. **Modules.** The functor may define a new name for an existing module passed as (part of) a functor argument. Within the functor, the sharing of the corresponding module definitions is known. Outside the functor, this sharing knowledge is hidden unless explicitly listed in the functor specification.

4. **Types.** The functor may define a new name for an existing datatype. Within the functor, this type definition is transparent i.e. the functor has always use the underlying representation. If this type name is contained in the specification of the functor

result, the type may be used outside the functor but its representation is hidden from the user.

All datatypes are abstract unless defined within the current functor. All types not defined within the current functor are passed as part of the functor arguments; there need not be any built-in types.

5. **Core Definitions.** The functor may contain value, function and procedure definitions in the core language. If there is a constructive definition of an item in the result specification of the functor, the corresponding core definition may be omitted.

   Core definitions may be declared as inline; in this case every occurrence of the defined name (also by any user of the functor result) may be textually replaced by its definition provided that instantiated code is generated for the functor application.

Another conceptual part of a functor is also the proof that the functor result meets the properties of the result specification. In this project we do not deal with the proofing aspect of functors. However, the necessary prerequisites are all there (formal knowledge in specifications) to incorporate also formal proofs into functors.

5.4.1 Predefined Instantiations

The implementation of abstract datatypes has to be based on a set of pre-defined modules that represent the interface to the hardware architecture and to the operating system respectively.

For instance, we have a module with specification

```
spec Proc = {

  type Word
  module Bool: Bool
  type Bool = Bool.Bool

  fun equal(Word, Word): Bool
  ...

  fun sum(Word, Word): Word
  ...

}
```

that defines the datatype "machine word" with the corresponding primitive arithmetic operations. For expressing equality, this specification is based on a `Bool` module that gives us access to the conditional operation of the machine language.

The mathematical approach of describing processor (finite field) arithmetic, is based on the concept of integer arithmetic. This requires an `Integer` module in `Proc` to express the `Proc` operations by `Integer` operations. However, on actual computers, the situation is the other way round: initially we are just given finite field operations on top of which arbitrary precision arithmetic can be implemented. Since a module that implements a specification must also implement all submodules of the specification, this approach is not feasible. The only solution is to follow the hardware implementation by describing the bit-algorithms of some fundamental operations and to express all other operations on top of them.

The most essential part of the operating system interface is the access to the standard input/output stream:

```
spec IO = {

  module Proc: Proc

  type Word = Proc.Word
  type Address = Proc.Address

  proc write(Address, Word)
  proc read(Address, Word)
  ...

}
```

However, for formally describing the concept of input/output the specification language is too weak. In particular, we are not able to give operations that read the output stream and thus cannot describe the effect of `write` by user-defined functions.

Likewise, we need access to the system heap

```
spec Heap = {

  module Proc: Proc

  type Word = Proc.Word
  type Address = Word

  fun alloc(Word): Address
  proc free(Address)

}
```

For allocating objects on the heap, the corresponding type must export its representation size as follows:

```
spec SizedType = {
  module Proc: Proc
  module Type: EqType
  type Type = Type.Type
  val size: Proc.Word
}
```

Furthermore we need a functor that implements typed pointers as follows:

```
spec Pointer[Elem:SizedType] = {

  module Proc:Proc

  type Pointer
  type Type = Pointer
  type Elem = Elem.Type

  val size: Word
  proc pointer(Word): Pointer
  fun read(Pointer): Elem
  proc write(Pointer, Elem)

}
```

These and other modules can be combined to an integrated system interface

```
spec System = {
  module Proc: Proc
```

```
    module Heap: Heap
    ...
    prop Proc = Heap.Proc
    ...
    import Proc
    import Heap
    ...
  }
```

on top of which standard implementations of the fundamental program types and type constructors can be constructed.

5.4.2 Type Modelling

The implementation of functors corresponding to the types and type constructors described in Section 5.3 does not raise many fundamental problems. The only more difficult point is the implementation of the variable-argument functors for product and sum types.

The following functor sketches the implementation of such a product functor where tuples are allocated on the heap and represented as heap addresses:

```
  inline functor Product[ S: System, L:<< (<I>, M:SizedType) >> ]: ... =

    import S
    type Product = Heap.Address

    L(E):<<
      import Pointer(E.M)
    >>

    inline fun product( L(E):<< E.IDENT(x): E.M.Type >> ): Product = {
      size: Word = L.FOLD(Proc.zero):<<Proc.sum, M.size>>
      p: Product = alloc(size)
      L(E):<<
        offset: Word = E.PREFIX(Proc.zero):<<Proc.sum, E.M.size>>
        write(pointer(p+offset), E.x)
      >>
      return(p)
    }

    L(E):<<
      inline fun component<E.I>(p: Product): E.M.Type =
      {
        offset: Word = E.PREFIX(Proc.zero):<<Proc.sum, E.M.size>>
        return(read(pointer(p+offset)))
      }
    >>

  }
```

In this functor, the term

E.IDENT(x)

generates a new identifier for every component E of L; this identifier may be subsequently referenced as $E.x$.

The term

E.FOLD(z)<<f, e>>

denotes $f(e_1, f(e_2, \ldots f(e_n, z)))$ for the argument list L of length n. Likewise, for every prefix list L_i

E.PREFIX(z)<<f, e>>

denotes $f(e_1, f(e_2, \ldots f(e_i, z)))$. With these expressions, we can compute in a portable way the total size of the heap object and the address offsets of the individual product components.

The `inline` annotations of the constructor/selector functions denote that all applications of these functions should by textually replaced by the corresponding function bodies without argument evaluations and function calls actually taking place *provided them instantiation code is generated for the result module of the functor.* Annotating the functor declaration itself as `inline` denotes that by default for the result module of the functor instantiation code is to be generated (see Subsection 5.6). By these annotations, the performance of user-modelled type constructions is the same as that of corresponding built-in types.

5.4.3 Application Functors

At a higher level of the hierarchy, functors construct large-scale components of application programs that mainly interact with semantically related modules on the same level of abstraction. Most of these functors share the same set of fundamental datatypes and type constructors such that it would be much too tedious to represent each of them by an individual functor parameter.

As a rule of thumb, program functors are only parameterized by modules that reside on the same level of abstraction. All lower-level modules on which a functor depends are typically grouped into a single functor parameter that represents the "language" used for implementing the functor.

Such a module thus replaces the concept of "built-in" types and operations, for instance:

```
spec Standard = {

  module Bool: Bool
  module System: System
  module Integer: Integer
  module Char: Char
  ...

  functor Array[EqType]: Array
  functor Product[ ... ]: ...
  ...

  prop Bool = Integer.Bool
  ...

  import Bool
  ...

}
```

A high-level program functor will then have a structure like

```
functor Compiler[Std:Standard, P:Parser, L:LexAn]: Compiler = {

  import Std
  prop P.Std = Std
  prop L.Std = Std
```

```
    ...
  }
```

where the standard types are imported into the current environment such that their use does not at all differ from datatypes built-in into the core language.

However, unlike built-in types the implementation of modules can be changed by the programmer at any time without difference for the application. Independent modules can use different implementations of the same specification within a single application. There are no "hard-wired" imports within modules; all interaction is done by functor arguments that can be replaced at any time.

5.5 Instantiations

The final target of a functor library is the construction of a fully instantiated module that can be exported into a library of the given core language and compiled by the native compiler (for instance to a Unix object file archive with the corresponding header file).

In the proposed system, the actual creation of modules is only triggered by top-level instantiations like

```
module System: System =
  ...

module StdInt: Integer =
  let module Y: System = System
      module B: Bool = BoolFunct[inline Y]
      module S: Sequence = SeqFunct[Y,B]
      ...
  in IntFunct[inline Y, inline B, S, ...]
```

Every instantiation constructs a module by the application of a functor to modules constructed from other instantiations (global or local ones). A functor argument may be tagged as `inline` which denotes that all references to functions/procedures exported by the module should be resolved at compile-time. The tag `noinline` for a functor argument overrides the `inline` tag of the corresponding functor parameter. The effects of these tags are described in the next section in more detail.

5.6 Implementation

The module system will be implemented as an **open library** with precise specifications and clear separation from the interface to the core language. This language-dependent interface performs the task of parsing textual input into abstract syntax trees, manipulating these trees, and finally generating from these trees textual output for the core compiler. In the frame of the current project, one such interface (to a subset of the C language) will be provided. In continuation projects tools may be developed that automatically generate these interfaces from formal language definitions.

We will build the library implementing the module system in such a style that it can be itself compiled using this module system. The library will be therefore implemented in a subset of C augmented by annotations that represent a functor interface but are ignored by the core C compiler during the bootstrapping phase.

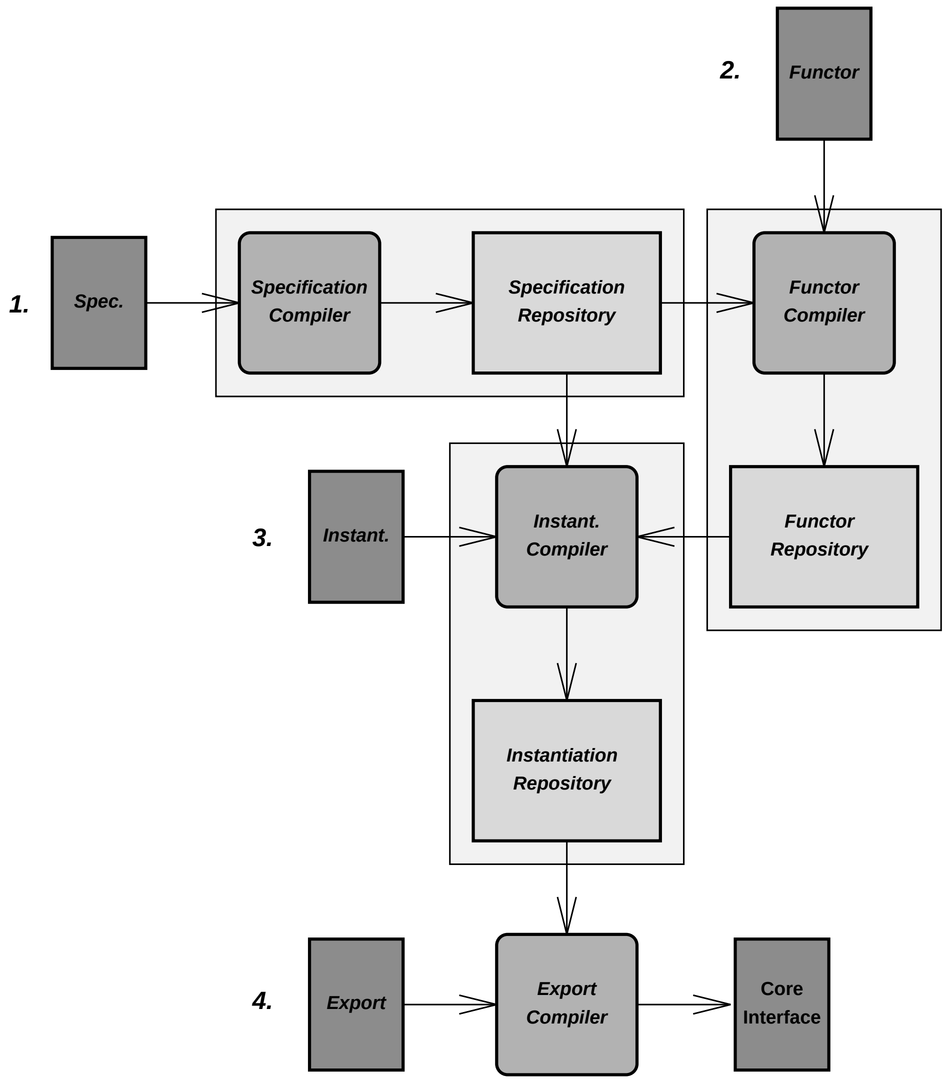

Figure 5.2: System Structure

5.6.1 System Structure

The general structure of the compilation system is outlined in Figure 5.2:

1. **Specifications** are compiled and collected in a specification repository.

2. **Functors** are compiled using the contents of the specification repository; the generated abstract code is collected in a functor repository.

3. **Instantiations** are compiled using the contents of the specification and of the functor repository; the generated instantiation code is collected in an instantiation repository.

4. **Exports** generate interface definitions for use of the instantiation repository from application programs in the core language.

Each repository is itself structured according to the namespace structure of the library.

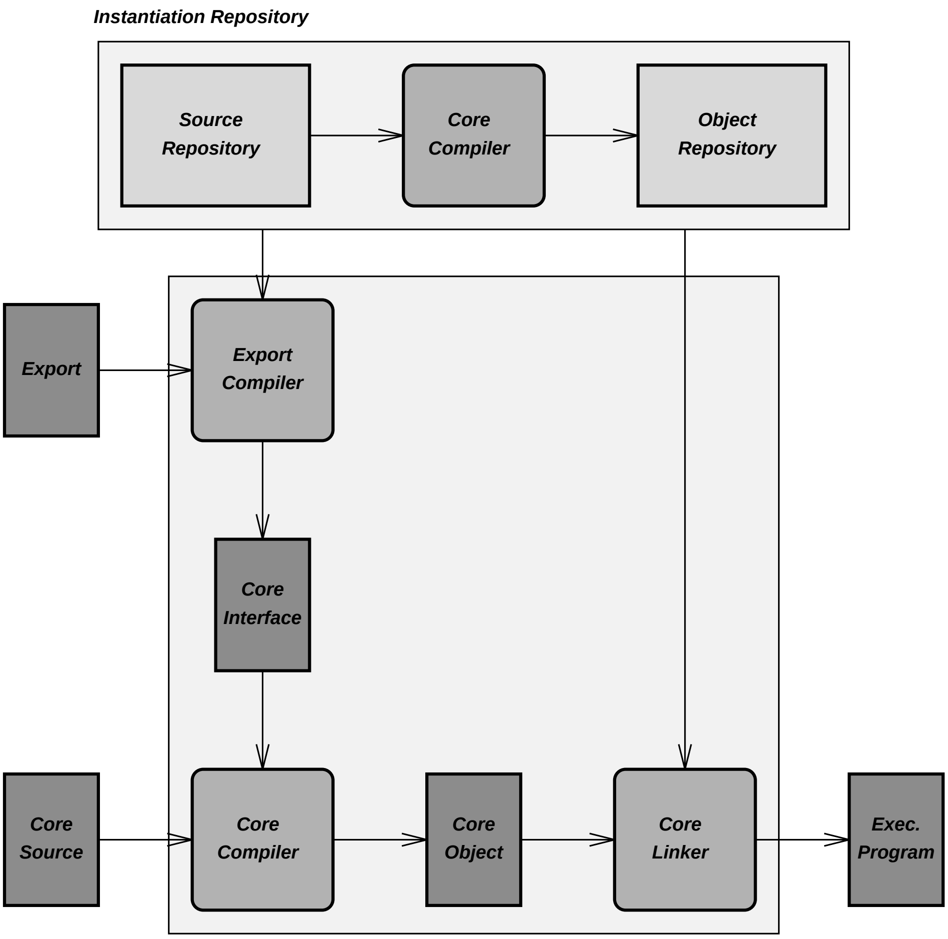

Figure 5.3: System Application

Core Language

The relationship of the system to the core language is illustrated in more detail in Figure 5.3:

1. The instantiation repository contains the core language source code of the generated functor instantiations. This source code is compiled by the core language compiler into object files that are collected in archives.

2. An application program can use the interface definition generated by the export compiler to refer to the contents of the instantiation repository; the core language compiler generates object files with the corresponding external references.

3. The object files are linked by the native linker with the required object file archives of the instantiation repository to generate an executable program.

File Dependencies

The dependencies between different files (source and object code) are maintained in the following way:

1. Within the library, the system automatically keeps track of all dependencies among specifications, functors, and instantiations. When an export request is issued, all necessary files are compiled automatically without requiring explicit user actions.

2. Outside the library, the user is responsible for maintaining the dependencies between application code in the core language and the contents of the library. However, the export compiler assists the user by providing all necessary information for standard dependence management tools (e.g. `make`).

It is important to note that export requests issued by different users do not duplicate instantiation code. Instantiation code whose generation is triggered by one user is also available for other users. Thus the total size of object file archives in a system is kept at a minimum.

Package Protection

For using the library, we must distinguish two levels of protection that are associated to every library package:

1. **Development.** Developers of a library package are allowed to add and to modify specifications, functors, and instantiations.

2. **Application.** Users of a library package are not allowed to modify the contents of the library but may still generate instantiation code.

Every namespace may be protected accordingly by giving or denying the right to modify and to use the package, respectively.

5.6.2 Compilation

A central part of the system is the **functor compiler** whose main tasks are the following:

Error Checking

Every functor implements a specification consisting of the specification of the functor arguments and of the specification of the functor result. The compiler can therefore check the correctness of the functor without knowledge of any other functor. This includes the check for type correctness (i.e. whether the interface of the functor matches its core implementation) but, in principle, also the check for semantic correctness (i.e. whether the implementation fulfills the properties listed in the specification).

In the current project, we will *not* investigate automatic program verification but rely on the programmer who justifies for himself the correspondence of the implementation with the specification.

Anyway, we already consider the possibility to formally express the behavior of a function *within* the program as an important fact, because it represents a "contract": any implementation that violates the specification can be considered illegal, even if the violation is not automatically detected.

We allow both the compiler and the user to rely on these laws, the former for generating code, the later for applying this code. Actually we consider more important than program verification is the aspect of *falsification*: anybody can falsify an implementation by showing how it violates for certain inputs some law of the specification.

Overloading

As a necessary prerequisite of error checking, the compiler has to resolve the overloading of (in particular function) names: for every occurrence of a name n in a functor or specification, the compiler has to decide whether n is defined locally within a block of the core language. If this is not the case, n has to be replaced by a *path expression* $M_0.M_1.\dots.n$ where the source module of the corresponding object is explicitly denoted (M_0 is either a formal functor parameter or the result of the functor application). Different functions may have the same name only if their interfaces differ in at least one argument position. Function overloading can therefore be simply resolved on the basis of the argument types; the target code will contain a unique name for every individual function.

Default Implementations

The specification of a functor result may describe some of the exported components such that the compiler may automatically generate a constructive definition of the component if the functor does not itself contain a definition for that component. In this way, duplicate definitions (in the specification and in the functor) may be avoided but the programmer is free to replace definitions by more efficient ones.

In particular, a functor may contain an imperative definition of a functor component while its specifications is always declarative.

Result

Every functor can therefore be translated into a compilation unit of the core language where every entity declared on the top-level has a unique name, a definition in the core language and all references to non-local names are made explicit. How from this form actual code is generated, is described in the next subsection.

5.6.3 Code Generation

For the code generation of the functor-based system there basically exist two strategies that correspond to the decision whether the `functor compiler` or the `instantiation compiler` perform this task.

1. **Run-Time Application.** Every functor may be compiled independently into target code. At runtime, functors are just functions that take modules as arguments and return modules as results. The runtime representation of every module is a table whose slots contain pointers to the exported functions.

   The advantage of this approach is that code is generated for every functor only once which reduces the compilation time and limits the code size.

2. **Compile-Time Instantiation.** For every application of a functor, target code may be generated. All references to components of argument modules are resolved at compile-time i.e. every module is just a piece of code without any representation as a runtime object.

   The advantage of this approach is that there is no runtime overhead for accessing the components of argument modules; moreover that the code of these components can be in principle inlined into the caller.

We propose the integration of both approaches where code explosion can be avoided but compile-time resolution of references and inlining is possible whenever it is considered as crucial for the performance:

1. **Default.** When a functor is compiled, always closed code is generated by representing the functor as a runtime function that takes modules as its arguments and returns a module as its result. Consequently every functor application $F[M_i]$ is by default compiled into a function application.

2. **Inlining of Modules.** This default implementation may be overridden by instantiations because they are declared at the top-level (i.e. there is no enclosing functor) and thus in every application $F[M_i]$ both F and M_i are known:

   If the occurrence of M_i in the application is tagged as `inline`, an instantiation of F is produced where all references to components of M_i are resolved at compile-time. Likewise, if the whole application is tagged as `inline`, references to components of the application result are in all occurrences resolved at compile-time. If local functor applications (within other functors) are tagged correspondingly *and* the functors and functor arguments become known (by the top-level instantiations), compile-time resolution may proceed recursively.

   The same instantiations may be also triggered by `inline` tags in functor definitions (for the formal parameters and/or the functor result) unless overridden by `noinline` tags in the actual applications.

3. **Inlining of Functions.** If a module $M = F[\ldots]$ is inlined in an application $G[M]$, the definition of a function f exported by M and tagged as `inline` in F may be inlined for every application of f in G (provided that the definition of f is not recursive). If f is not used as a first-order object, code need not be generated for f at all.

4. **Code Sharing.** The system keeps track of all instantiations generated by any user of the functor library such that code is generated at most *once* for every application of a functor to the same arguments, even if the instantiations are triggered by different compilation units or by different users.

5. **Exports.** For use in an application program, a module may be exported into the environment of the core language. The result of this export consists of interface definitions and of (references to) a a set of object file archives:

   (a) The interface definition represents the specification of the exported module translated into the notion of the core language (e.g. as a C header file). It also maps the names listed in the specification to the names used internally within the library (different modules may have the same specification). The programmer may also demand name prefixes for all exported items for the simultaneous use of different instantiations of a functor.

   (b) The object file archives contain the compiled code generated for all instantiations of all functors. These archives are located in a central repository that is shared by all users of the functor library. Any instantiation generated by one user may be immediately used by any other user without recompilation.

6. **Dependencies.** The system keeps track of all dependencies among modules, functors, specifications and the resulting target code. Whenever a user demands the export of a particular module, these dependencies are automatically resolved and the individual units are recompiled if necessary.

5.7 Future Prospects

Based on the result of this project, the development of various tools (based on existing software) will be pursued in future continuation projects:

- **Computer-Aided Documentation** tools for the automatic generation of reference manuals and on-line help documents for functor libraries.

- **Graphical Browsing** tools that support the management and development of functor libraries.
- **Debugging Support** that allows the use of existing source code debuggers in the context of generic source code.
- **Language Development** tools that allow the simple embedding of different core languages into the module system.

Using these tools **functor libraries** will be constructed for various application domains that will also provide valuable feedback for the further development of the system.

In particular we will in a continuation project pursue the porting of the STURM library for constraint solving [Hong *et al.*, 1994] from C++ to the proposed functor language.

Chapter 6

Project Schedule

This chapter from the original proposal is omitted.

Chapter 7

Researchers

7.1 The Institute

The Research Institute for Symbolic Computation (RISC-Linz) of the Johannes Kepler University Linz has pursued for almost two decades research on computer algebra, computational geometry, computational logic, automated reasoning, constraint solving, functional and logic programming, parallel and distributed processing with applications in geometric modelling, computer-aided engineering, robotics, NC-programming, expert systems, neural networks, etc. The institute whose chairman is Prof. Bruno Buchberger has participated in numerous national research projects (supported by Austrian Science Foundation FWF and the Austrian Federal Ministry of Science and Research BMWF), international programs (e.g. the European ESPRIT and the Japanese Real-World Computing programme) and collaborated in various development projects with industrial partners.

RISC-Linz has served as the host of numerous scientific workshops and of various international conferences, e.g. the EUROCAL '85 International Symposium on Symbolic and Algebraic Computation [Buchberger, 1985], the DISCO '93 International Symposium on the Design and Implementation of Symbolic Computation Systems [Miola, 1993], the PASCO '94 International Symposium on Parallel Symbolic Computation [Hong, 1994] and has been co-organizer of the CONPAR 94/VAPP VI International Conference on Parallel and Vector Processing [Buchberger and Volkert, 1994]. Members of the institute have been regularly invited to the program committees of large international conferences.

The institute has a faculty of about 15 members with doctoral degree. About 50 students from mathematics and computer science (more of half of which are from abroad) currently pursue their diplomas respectively Ph.D. Five doctoral students of RISC-Linz have graduated *sub auspiciis praesidentis*.

7.2 Generic Programming and Mathematical Software

From its background in symbolic and algebraic computation, researchers at RISC-Linz have also a long-standing experience in the development of mathematical software libraries and the associated issues of generic programming:

7.2.1 Software Libraries

In the past a variety of libraries of mathematical algorithms has been developed. Most of them were non-generic mostly because they were not adequately supported by the underlying software systems but also because the importance of genericity was at that time not yet fully understood.

In 1987 a geometry proving package [Heinzelreiter *et al.*, 1987, Kusche *et al.*, 1987] was developed for Scratchpad II, the predecessor of the AXIOM system from which the A# language was recently derived. The system was able to automatically prove theorems in 2D geometry as can be found in high-school textbooks.

The computer algebra library SACLIB [Buchberger *et al.*, 1992, Hong *et al.*, 1993] was developed in C to efficiently support arithmetic in various mathematical domains. Several large-scale application packages were developed on top of SACLIB. PACLIB [Schreiner and Hong, 1993a, Schreiner and Hong, 1993b], a parallel variant of the SACLIB runtime kernel, was developed in the frame of an FWF project for the parallelization of computer algebra algorithms on a shared memory multiprocessor.

The CASA package [Mnuk *et al.*, 1995, Winkler, 1993, Wall, 1991, Gebauer *et al.*, 1991] implements in Maple a number of algorithms for constructive algebraic geometry. The objects that CASA works with are algebraic sets in affine or projective space over a field. The geometric objects may be given in various different representations. CASA is able to analyze various properties of these sets; the user can also create 2- and 3-dimensional pictures of curves and surfaces.

7.2.2 Signatures for C++

Gerald Baumgartner, a former student of RISC-Linz working on type systems for computer algebra [Baumgartner and Stansifer, 1990] developed at Purdue University an extension of C++ where signatures were used to express class interfaces [Baumgartner and Russo, 1994b, Baumgartner and Russo, 1994a]. This extension represented a major step into the direction of an algebraic formulation of abstract datatypes in C++ and was included into an experimental version of the GNU C++ compiler. However these C++ signatures are (like virtual classes) implemented by lookup tables which is too inefficient for large-scale generic programming.

7.2.3 A Systems Survey

In 1990 Bruno Buchberger performed a survey evaluating the usefullness of a wide-range of programming languages and systems for generic programming. As a result, it turned out that no available system combined genericity with efficiency sufficient enough for production-quality computation. An implementation of Gröbner bases in Mathematica proved elegant but to inefficient for practical purposes [Buchberger, 1990, Buchberger, 1991]. [Windsteiger, 1990] consequently made a proposal to express genericity via an object-oriented programming style in the C language.

7.2.4 Generic Gröbner Bases

As one result of the survey, at RISC-Linz the generic Gröbner Bases package GRÖBNER [Windsteiger and Buchberger, 1993, Windsteiger, 1993] was developed on top of the SACLIB library [Buchberger *et al.*, 1992]. GRÖBNER was implemented in C; genericity was implemented by preprocessing the source code such that the package could be instantiated for various mathematical domains at compile-time.

7.2.5 Generic Mathematics

In 1993 Bruno Buchberger has started the formulation and presentation of the "tower of mathematics" in the form of a Mathematica notebook constructing mathematical domains by "functors" (Mathematica functions) that map modules (Mathematica records storing functions) into other modules. Using the Mathematicas interpreter of higher-order equational logic, with these structures not only computing but also a form of semi-automated reasoning is possible. Mathematica is very well suited for this purpose, but the interpretation mechanism is slow and not suitable for the actual execution of time-expensive programs. For such purpose, algorithms have to be externally implemented in C and linked to the Mathematica kernel.

7.2.6 The STURM Library

In 1993 Dr. Hoon Hong started at RISC-Linz a project to develop a software library STURM [Hong *et al.*, 1994] for solving systems of constraints. This work was based on his previous experience with the development of the C library SACLIB [Buchberger *et al.*, 1992]. Key issues of the STURM project were provably correct algorithms, genericity, and performance. An evaluation of available programming languages revealed that only C++ [Stroustrup, 1991, Stroustrup, 1994] provided the potential for developing efficient generic mathematical code; consequently this language was chosen for the implementation of the library.

While the STURM project has succeeded and the library will be soon released for public use, the developers had to undergo a lot of pain and suffering: the project revealed numerous semantic and pragmatic deficiencies of C++ that considerably hampered the project and lowered the productivity. In particular genericity turned out to be a major stumbling-block for the whole development process [Caprotti *et al.*, 1994]. As a whole, the project shed a disillusioning light on the available support for generic programming.

7.3 The Principal Researchers

In the following we give a short background description of the principal researchers of this project. Their complete curricula vitae are listed in the appendix.

7.3.1 Dr. Hoon Hong

Dr. Hong obtained his Master's degree in physics and his Ph.D. in computer algebra (thesis title: "Efficient Quantifier Elimination Algorithm and Software for the First Order Theory of Real Closed Fields") in 1990 at the Ohio State University in the USA. Since then he is an assistant professor at RISC-Linz. He has published over 30 scientific papers in international conferences/journals, mostly in the area of quantifier elimination, numerical/interval mathematics, parallel symbolic computations, non-linear constraint solving and constraint logic programming systems.

He has edited two special issues of the journals "Annals of Mathematics and Artificial Intelligence" and of the "Computer Journal", respectively. Currently he is editing two more special issues of the "Journal of Symbolic Computation" and of the "Foundation of Computer Science", respectively. Dr. Hong is a nominee for the vice-chair of the ACM SIGSAM Special Interest Group for Symbolic and Algebraic Manipulation. He has organized the PASCO 94 International Conference on Parallel Symbolic Computation [Hong, 1994] that took place in 1994 in Linz, Austria. He has also participated in the program committees of various international conferences.

Dr. Hong was one of the principal developers of the computer algebra library SACLIB [Buchberger *et al.*, 1992] and of its shared-memory parallel variant PACLIB [Hong *et al.*, 1993]. He currently pursues a research project supported by the FWF, the Central European Initiative (CEI), and ESPRIT on the development of the constraint solving library STURM [Hong *et al.*, 1994] and its parallelization.

Dr. Hong is an expert in computer algebra, numerical mathematics, algorithm design, and mathematical software development. He has contributed to the overall design of the proposed functor system and will constantly evaluate the adequacy of the detailed design for the intended application domain. He will also direct the development of the basic functor library of container types and mathematical objects. In a continuation project, he will pursue the porting of the complete STURM library from C++ to the functor system.

7.3.2 Dipl-Ing. Dr.[1] Wolfgang Schreiner

Dr. Schreiner has worked at RISC-Linz since 1989 in the development of programming languages and software systems for parallel symbolic computation. In 1990 he graduated with his diploma thesis "ADAM and EVE — An Abstract Dataflow Machine and Its Programming Language" [Schreiner, 1990] for which he received the 1991 promotion award of the Austrian Computer Society (OCG) for the the best Austrian diploma thesis in computer science. Also in 1991 he received for excellent studies the appreciation award of the Austrian Federal Ministry for Science and Research.

Dr. Schreiner developed in collaboration with Dr. Hong the runtime kernel and programming interface of the parallel computer algebra library PACLIB [Schreiner and Hong, 1993a, Schreiner and Hong, 1993b] and built a para-functional programming language on top of this kernel [Schreiner, 1994a, Schreiner, 1995]. In December 1994 he successfully defended his Ph.D. thesis on "Parallel Functional Programming for Computer Algebra" [Schreiner, 1994b]. He has been granted graduation *sub auspiciis praesidentis* which is scheduled for November 1995. Dr. Schreiner has published about 10 scientific papers in international conferences and journals mostly on parallel symbolic computation and the parallelization of functional languages. He was one of the local organizers of the CONPAR 94/VAPP VI International Conference on Parallel and Vector Processing [Buchberger and Volkert, 1994] that took place in September 1994 at the University of Linz. He has been member of the program committee of the 1995 PVM European User's Group Meeting.

Besides his research in (sequential and parallel) systems for symbolic computation and the compilation of functional languages for parallel computers, Dr. Schreiner is an expert in the design and semantics of programming languages, type and module systems, construction of compilers and runtime systems. He will direct the detailed design of the functor system (with the feedback of Dr. Hong), the formalization of its semantics, and its implementation by compile-time code instantiation and by run-time function application.

7.4 The Student Coworkers

The student coworkers of this project will be chosen in time from the large supply of Ph.D. students graduating at RISC-Linz; all of them have a strong background in mathematics and in the formal foundations of computer science as well as in practical software development. The curricula vitae of the most promising candidates are given in the appendix.

[1] Graduation *sub auspiciis praesidentis* is pending.

Bibliography

[Akerkar *et al.*, 1994] Rajendra Akerkar, Timofei Tcherkasov, Andreas Neubacher, Ovidiu Podisor, and Rosario Salamone. Namespaces for the STURM Library. STURM Project Internal Note STURM-94-004, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 1994.

[Anderson *et al.*, 1995] E. Anderson, Z. Bai, et al. *LAPACK User's Guide.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 2nd edition, 1995.

[Appel and MacQueen, 1994] Andrew W. Appel and David B. MacQueen. Separate Compilation for Standard ML. Technical Report CS-TR-452-94, Princeton University, March 1994.

[Appel, 1993] Andrew W. Appel. A Critique of Standard ML. *Journal of Functional Programming*, 3(4):391–429, October 1993.

[Augustsson, 1987] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Baker, 1995] Henry G. Baker. 'Use-Once' Variables and Linear Objects — Storage Management, Reflection and Multi-Threading. *SIGPLAN Notices*, 30(1):45–52, January 1995.

[Barnes, 1994] J. Barnes. *Programming in Ada.* Addison-Wesley, Reading, MA, 4th edition, 1994.

[Baumgartner and Russo, 1994a] Gerald Baumgartner and Vincent F. Russo. Implementing Signatures for C++. In *1994 USENIX C++ Conference*, pages 47–56, Cambridge, Massachusetts, April 11–14, 1994.

[Baumgartner and Russo, 1994b] Gerald Baumgartner and Vincent F. Russo. Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism. Technical Report CSD-TR-93-059, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, December 1994, To appear in Software Practice and Experience, 1995.

[Baumgartner and Stansifer, 1990] Gerald Baumgartner and Ryan Stansifer. A Proposal to Study Type Systems for Computer Algebra. Technical Report 90-07.0, RISC-Linz, Johannes Kepler University, Linz, Austria, March 1990.

[Berry, 1993] Dave Berry. Lessons from the Design of a Standard ML Library. *Journal of Functional Programming*, 3(4):527–552, October 1993.

[Birtwistle *et al.*, 1979] Graham Birtwistle, Ole-Johan Dahl, Björn Myrhaug, and Kristen Nygaard. *SIMULA BEGIN.* Studentlitteratur, Lund, Sweden, 1979.

[Bruce, 1994] Hillam Bruce. *Introduction to Abstract Data Types Using Ada.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Buchberger and Volkert, 1994] Bruno Buchberger and Jens Volkert, editors. *Parallel Processing: CONPAR 94 — VAPP VI International Conference on Parallel and Vector Processing*, volume 854 of *Lecture Notes in Computer Science*, Linz, Austria, September 6–8, 1994. Springer, Berlin.

[Buchberger *et al.*, 1992] Bruno Buchberger, George Collins, Mark Encarnación, Hoon Hong, Jeremy Johnson, Werner Krandick, Rüdiger Loos, Ana Mandache, Andreas Neubacher, and Herbert Vielhaber. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, June 1992.

[Buchberger, 1985] Bruno Buchberger, editor. *EUROCAL '85 — International Symposium on Symbolic and Algebraic Computation*, volume 203–204 of *Lecture Notes in Computer Science*, Linz, Austria, April 1–3, 1985. Springer, Berlin.

[Buchberger, 1990] Bruno Buchberger. An Implementation of Gröbner Bases in Mathematica. Technical Report 90-58, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990.

[Buchberger, 1991] Bruno Buchberger. Gröbner Bases in Mathematica: Enthusiasm and Frustation. In *Proceedings IFIP TC2/WG2.5 Working Conference on Programming Environments for High-Level Scientific Problem Solving*, Karlsruhe, Germany, September 23–27, 1991. Also Technical Report 91-11, RISC-Linz, Johannes Kepler University, Linz, Austria, 1991.

[Caprotti *et al.*, 1994] O. Caprotti, H. Hong, D. Jakus, O. Podisor, S. Ratschan, and V. Stahl. On C++ Polymorphism for the STURM Library. STURM Project Internal Note STURM-94-001, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 1994.

[Cardelli and Wegner, 1985] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Char *et al.*, 1991] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan, and S. M. Watt. *Maple V Language Reference Manual.* Springer, New York, 1991.

[Cook *et al.*, 1990] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not Subtyping. In *Seventeenth Annual ACM Symposium onn Principles of Programming Languages*, pages 125–135, San Francisco, California, January 17–19, 1990. ACM Press, New York.

[Cousineau *et al.*, 1985] G. Cousineau, P. L. Curien, and M. Mauny. The Categorical Abstract Machine. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64, Nancy, France, September 16–19, 1985. Springer, Berlin.

[Cregut and MacQueen, 1994] Pierre Cregut and David B. MacQueen. An Implementation of Higher Order Functors. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 13–31, Orlando, Florida, June 25–26, 1994. ACM, New York.

[Dongarra *et al.*, 1978] J. Dongarra, , J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK User's Guide.* SIAM Publications, Philadelphia, PA, 1978.

[Futatsugi *et al.*, 1985] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *12th Annual Symposium on Principles of Programming Languages*, pages 52–66, New Orleans, LA, January 1985. ACM Press, New York.

[Gebauer *et al.*, 1991] R. Gebauer, Kalkbrenner M., B. Wall, and F. Winkler. CASA: A Computer Algebra Package for Constructive Algebraic Geometry. In S. M. Watt, editor, *ISSAC'91 — International Symposium on Symbolic and Algebraic Computation*, Bonn, Germany, July 1991. Also Technical Report 91-17, RISC-Linz, Johannes Kepler University, Linz, Austria, 1991.

[Girard, 1987] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Goguen and Winkler, 1988] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report CSL-88-9, SRI International, 1988.

[Gunter and Mitchell, 1993] C. Gunter and J. Mitchell, editors. *Semantics of Object-Oriented Programming Languages.* MIT Press, Cambridge, MA, 1993.

[Hall *et al.*, 1994] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type Classes for Haskell. In Donald Sannella, editor, *ESOP '94 — 5th European Symposium on Programming Languages and Systems*, Lecture Notes in Computer Science, pages 241–256, Edinburgh, UK, April 11–13, 1994. Springer, Berlin.

[Hammond and Blott, 1989] K. Hammond and S. Blott. Implementing Haskell Type Classes. In Kei Davis and John Hughes, editors, *Proceedings of the 1989 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 266–286, Fraserburgh, Scotland, August 21 – 23, 1989. British Computer Society, Springer, London.

[Hammond and Hall, 1991] K. Hammond and C. Hall. A Natural Dynamic Semantics for Haskell (Draft). Technical report, Department of Computing Science, Glasgow University, UK, February 1991.

[Harper and Lillibridge, 1994] Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages.* ACM Press, New York, January 1994.

[Harper *et al.*, 1990] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-Order Modules and the Phase Distinction. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990. ACM Press, New York.

[Hearn and Schrüfer, 1993] Anthony C. Hearn and Eberhard Schrüfer. An Order-Sorted Approach to Algebraic Computation. In Alfonso Miola, editor, *DISCO '93 — International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 134–144, Gmunden, Austria, September 15–17, 1993. Springer, Berlin.

[Heinzelreiter *et al.*, 1987] Johann Heinzelreiter, Klaus Kusche, Bernhard Kutzler, and Herwig Mayr. A Geometry Theorem Proving Package in SCRATCHPAD II. In G. Pernul and A. Min Tjoa, editors, *Berichte aus Informatik Forschungsinstitutionen (10)*, pages 349–358, Vienna, Austria, February 25–27, 1987. Oldenburg.

[Hong *et al.*, 1993] Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner. The Design of the SACLIB/PACLIB Kernels. In Alfonso Miola, editor, *DISCO 93 — International Symposium on Design and Implementation of Symbolic Computation System*, volume 722 of *Lecture Notes in Computer Science*, pages 288 – 302, Gmunden, Austria, September 15–17, 1993. Springer, Berlin, Also: Technical Report 93-30, RISC-Linz, Johannes Kepler University, Linz, Austria, June 1993.

[Hong *et al.*, 1994] Hoon Hong, Andreas Neubacher, and Volker Stahl. The STURM Library Manual — A C++ Library for Symbolic Computation. Technical Report 94-30, RISC-Linz, Johannes Kepler University, Linz, Austria, April 1994.

[Hong, 1994] Hoon Hong, editor. *PASCO 94 International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 26–28, 1994. World Scientific Publishing, Singapore.

[Hudak *et al.*, 1992] Paul [editor] Hudak, Simon [editor] Peyton Jones, Philip [editor] Wadler, Brian Boutel, Jon Fairbairn, Joeseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. Research Report 1.2, Department of Computer Science and Department of Computing Science, Yale University, New Haven, Conneticut, U.S.A. and University of Glasgow, Glasgow, U.K., March 1992.

[Huet, 1990] G. Huet, editor. *Logical Foundations of Functional Programming.* University of Texas Year of Programming Series, Addison-Wesley, Reading, MA, 1990.

[Jenks and Sutor, 1992] R. Jenks and R. Sutor. *AXIOM — The Scientific Computation System.* Springer, New York, 1992.

[Johnsson, 1987] Thomas Johnsson. *Compiling Lazy Functional Languages, Part I.* PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Kahn and others, 1980] G. J. Kahn et al. Formal Definition of the Ada Programming Language. Technical report, Institut National de Recherche en Informatique et en Automatique, France, 1980.

[Kahrs *et al.*, 1993] Stefan Kahrs, Donald Sannella, and Andrej Tarlecki. The Semantics of Extended ML: A Gentle Introduction. In *International Workshop on Semantics of Specification Languages*, Springer Workshops in Computing, pages 186–215, Utrecht, 1993.

[Kahrs *et al.*, 1994] Stefan Kahrs, Donald Sannella, and Andrej Tarlecki. The Definition of Extended ML. Technical Report ECS-LFCS-94-300, Laboratory for Computer Science, University of Edinburgh, August 1994.

[Kando, 1993] Kando. A Collection of C++ Classes for Computer Algebra. Technical report, Fujitsu, 1993.

[Kernighan and Ritchie, 1988] Brian Kernighan and Dennis Ritchie. *The C Programming Language (2nd edition).* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Kusche *et al.*, 1987] Klaus Kusche, Bernhard Kutzler, and Herwig Mayr. Implementation of a Geometry Theorem Proving Package in SCRATCHPAD II. In J. Davenport, editor, *EUROCAL '87, Proceedings of the International Symposium on Symbolic and Algebraic Computation*, volume 378 of *Lecture Notes in Computer Science*, pages 246–257, Leipzig, GDR, June 2–5, 1987. Springer, Berlin.

[MacQueen and Tofte, 1994] David B. MacQueen and Mads Tofte. A Semantics for Higher-Order Functors. In Donald Sannella, editor, *ESOP '94 — 5th European Symposium on Programming Languages and Systems*, Lecture Notes in Computer Science, pages 409–423, Edinburgh, UK, April 11–13, 1994. Springer, Berlin.

[MacQueen, 1984] D. MacQueen. Modules for Standard ML. In *ACM Conference on Lisp and Functional Programming*, pages 198–207, Austin, Texas, 1984. ACM Press, New York.

[MacQueen, 1986] David MacQueen. Using Dependent TYpes to Express Modular Structures. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 277–286. ACM Press, New York, January 1986.

[McCarthy, 1960] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Communications of ACM*, 3(4):184–195, April 1960.

[Mehlhorn and Näher, 1992] Kurt Mehlhorn and Stefan Näher. Algorithm Design and Software Libraries: Recent Developments in the LEDA project. In *Algorithms, Software, Architectures — Information Processing 92*, volume 1, Amsterdam, 1992. Elsevier North-Holland.

[Mehlhorn and Näher, 1995] Kurt Mehlhorn and Stefan Näher. LEDA: A Platform for Combinatorial and Geometric Computing. *Commmunications of the ACM*, 38(1):96–102, January 1995.

[Meyer, 1988] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Meyer, 1991] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Milner *et al.*, 1990] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[Milner, 1984] R. Milner. A Proposal for Standard ML. In *ACM Conference on Lisp and Functional Programming*, pages 184–197, Austin, Texas, 1984. ACM Press, New York.

[Miola, 1993] Alfonso Miola, editor. *DISCO '93 — International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, Gmunden, Austria, September 15–17, 1993. Springer, Berlin.

[Mitchell and Harper, 1988] John C. Mitchell and Robert Harper. The Essence of ML. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 28–46. ACM Press, New York, January 1988.

[Mnuk *et al.*, 1995] Michal Mnuk, Bernhard Wall, and Franz Winkler. CASA Reference Manual (Version 2.2). Technical Report 95-05, RISC-Linz, Johannes Kepler University, Linz, Austria, 1995.

[Monagan, 1993] Michael B. Monagan. Gauss: a Parameterized Domain of Computation System with Support for Signature Functions. In Alfonso Miola, editor, *DISCO '93 — International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 81–94, Gmunden, Austria, September 15–17, 1993. Springer, Berlin.

[Naiditch, 1995] D. J. Naiditch. *Rendezvouz with Ada 95.* John Wiley & Sons, New York, 1995.

[Nelson, 1991] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Peyton Jones and Wadler, 1991] Simon L. Peyton Jones and Philip Wadler. A Static Semantics for Haskell. Technical report, Department of Computing Science, Glasgow University, UK, May 1991.

[Rayna, 1987] G. Rayna. *REDUCE — Software for Algebraic Computation.* Springer, New York, 1987.

[Reynolds, 1990] J. C. Reynolds. Introduction to Polymorphic Lambda Calculus. In G. Huet, editor, *Logical Foundations of Functional Programming*, University of Texas Year of Programming Series, pages 77–86, Addison-Wesley, Reading, MA, 1990.

[Sannella and Tarlecki, 1991] Donald Sannella and Andrej Tarlecki. Extended ML: Past, Present, and Future. Technical Report ECS-LFCS-91-138, Laboratory for Computer Science, University of Edinburgh, 1991.

[Santas, 1993] Philip S. Santas. A Type System for Computer Algebra. In Alfonso Miola, editor, *DISCO '93 — International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 177–191, Gmunden, Austria, September 15–17, 1993. Springer, Berlin.

[Schmidt, 1994] David A. Schmidt. *The Structure of Typed Programming Languages.* Foundations of Computing Series, MIT Press, Cambridge, MA, 1994.

[Schreiner and Hong, 1993a] Wolfgang Schreiner and Hoon Hong. A New Library for Parallel Algebraic Computation. In Richard F. Sincovec et al., editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume II, pages 776–783, Norfolk, Virginia, March 22–24, 1993. SIAM, Also: Technical Report 92-73, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992. Also: Technical Report ACPC/TR 93-9, ACPC Technical Report Series, Austrian Center for Parallel Computation, February 1993.

[Schreiner and Hong, 1993b] Wolfgang Schreiner and Hoon Hong. PACLIB — A System for Parallel Algebraic Computation on Shared Memory Computers. In Hussein M. Alnuweiri, editor, *Parallel Systems Fair at the Seventh International Parallel Processing Symposium*, pages 56–61, Newport Beach, CA, April 14, 1993. IPPS '93, Also: Technical Report 93-03, RISC-Linz, Johannes Kepler University, Linz, Austria, February 1993. Also: Technical

Report ACPC/TR 93-12, ACPC Technical Report Series, Austrian Center for Parallel Computation, February 1993.

[Schreiner, 1990] Wolfgang Schreiner. ADAM & EVE — An Abstract Dataflow Machine and Its Programming Language. Master's thesis, Johannes Kepler University, Linz, Austria, October 1990, Also: Technical Report 90-42.0, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990. Also: Technical Report 91-1, Austrian Center for Parallel Computation, January 1991.

[Schreiner, 1994a] Wolfgang Schreiner. A Para-Functional Programming Interface for a Parallel Computer Algebra Package. In Hoon Hong, editor, *PASCO 94 International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 26–28, 1994. World Scientific Publishing, Singapore.

[Schreiner, 1994b] Wolfgang Schreiner. *Parallel Functional Programming for Computer Algebra*. PhD thesis, Johannes Kepler University, Linz, Austria, September 1994, RISC-Linz Report 94-61.

[Schreiner, 1995] Wolfgang Schreiner. Application of a Para-Functional Language to Problems in Computer Algebra. In Wim Böhm and John Feo, editors, *HPFC '95 Conference on High Performance Functional Computing*, Denver, Colorado, April 9–11, 1995. Lawrence Livermore National Laboratories.

[Scratchpad, 1988] Computer Algebra Group, Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY. *The Scratchpad II Computer Algebra System – Interactive Environment Users Guide, Draft 1.2*, September 1988.

[Shao and W., 1993] Zhong Shao and Appel Andrew W. Smartest Recompilation. Technical Report CS-TR-395-92, Princeton University, January 1993.

[Stallman, 1992] Richard M. Stallman. Using and porting gnu cc (v. 2.3). Technical report, Free Software Foundation, December 1992.

[Standard, 1983] Military Standard. Reference Manual for the Ada Programming Language. Technical Report ANSI/MIL-STD-1815A-1983, United States Government Printing Office, 1983.

[Standard, 1994] International Standard. Ada Reference Manual — Language and Standard Libraries (Version 6.0). International Standard ISO/IEC 8652:1995(E), International Organization for Standardization, December 1994.

[Steele Jr., 1984] G. L. Steele Jr. *Common Lisp: The Language*. Digital Press, Burlington, MA, 2nd edition, 1984.

[Stroustrup, 1988] Blarne Stroustrup. Parameterized Types for C++. In *USENIX C++ Conference*, Denver, Colorado, October 1988. Also: USENIX Computer Systems, Vol. 2, No. 3, Winter 1989.

[Stroustrup, 1991] Blarne Stroustrup. *The C++ Programming Language (2nd edition)*. Addison-Wesley, Reading, MA, 1991.

[Stroustrup, 1994] Blarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

[Taft, 1992] S. Tucker Taft. Ada 9X: A Technical Summary. *Communications of the ACM*, 35(11), November 1992.

[Tofte, 1992] Mads Tofte. Principal Signatures for Higher-Order Program Modules. In *Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 189–199. ACM Press, New York, January 1992.

[Wadler and Blott, 1989] P. Wadler and S. Blott. How to Make *ad hoc* Polymorphism Less *ad hoc*. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, January 1989. ACM Press, New York.

[Wadler, 1991] Philip Wadler. Is There a Use for Linear Logic? In *ACM/IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation (PEPM)*, Yale University, June 1991.

[Wall, 1991] Bernhard Wall. CASA: Computer Algebra Software for Constructive Algebraic Geometry in MAPLE. A Primer. Technical Report 91-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1991.

[Watt *et al.*, 1988] Stephen M. Watt, Richard D. Jenks, Robert S. Sutor, and Barry M. Trager. The Scratchpad II Type System: Domains and Subdomains. Technical report, Computer Algebra Group, Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, 1988.

[Watt *et al.*, 1994] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. A# User's Guide (Version 0.37.0). Technical report, The Numerical Algorithms Group, September 1994.

[Weber, 1993] Andreas Weber. On Coeherence in Computer Algebra. In Alfonso Miola, editor, *DISCO '93 — International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 95–106, Gmunden, Austria, September 15–17, 1993. Springer, Berlin.

[Windsteiger and Buchberger, 1993] Wolfgang Windsteiger and Bruno Buchberger. GRÖBNER: A Library for Computing Gröbner Bases based on SACLIB. Technical Report 93-72, RISC-Linz, Johannes Kepler University, Linz, Austria, 1993.

[Windsteiger, 1990] Wolfgang Windsteiger. An Approach to Object-Oriented Programming in C. Technical Report 90-57, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990.

[Windsteiger, 1993] Wolfgang Windsteiger. Using GRÖBNER as a "Black Box". Technical Report 93-71, RISC-Linz, Johannes Kepler University, Linz, Austria, 1993.

[Winkler, 1993] Franz Winkler. Constructive Algebraic Geometry with CASA. In *COCOA III*, Cortona, Italy, May 31 – June 4, 1993. Also: Technical Report 93-26, RISC-Linz, Johannes Kepler University, Linz, Austria.

[Wirth, 1985] Niklaus Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science, Springer, Heidelberg, third, corrected edition, 1985.

[Wolfram, 1991] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, Redwood City, CA, 1991.

[Zippel, 1993] Richard Zippel. The Weyl Computer Algebra Substrate. In Alfonso Miola, editor, *DISCO '93 — International Symposium on the Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 303–318, Gmunden, Austria, September 15–17, 1993. Springer, Berlin.

Appendix A

Curricula Vitae