Programming with Structures, Functions, and Objects*

— Preliminary Version —

Fritz Henglein
DIKU
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen
Denmark
henglein@diku.dk

Konstantin Läufer
Courant Institute of Mathematical Sciences
New York University
715 Broadway, 7th floor
New York, NY 10003
USA
laufer@cs.nyu.edu

April 22, 1991

Abstract

We describe program structuring mechanisms for integrating algebraic, functional and object-oriented programming in a single framework. Our language is a statically typed higher-order language with specifications, structures, types, and values, and with universal and existential abstraction over structures, types, and values.

We show that existential types over structures generalize both the necessarily homogeneous type classes of Haskell and the necessarily heterogeneous object classes of object-oriented programming languages such as C++ or Eiffel. Following recent work on ML, we provide separate linguistic mechanisms for reusing specifications and structures. Subtyping is provided in the form of explicit type conversions.

The language mechanisms are introduced by examples to emphasize their pragmatic aspects. We compare them with the mechanisms of XML+, Haskell and Eiffel and give a type-theoretic perspective. These mechanisms have been developed within a larger, ongoing prototyping language design project.

1 Introduction

We describe program structuring mechanisms for a high-level language currently under development as part of a software prototyping research effort. Our main aim is to support and integrate algebraic, functional, and object-oriented programming in a coherent language framework.

The language concepts described in this paper are intended to serve as a platform for the design of the prototyping language Griffin [D+90] currently under development at New York University.

Our linguistic mechanisms can be seen as a combination of Ada (generic) packages [Uni83], ML polymorphism, signatures and structures [MTH90], Eiffel object classes [Mey88], Haskell type classes [HW90], and (partially) abstract types [MP88]. Our design is most closely related to the recently proposed extension of Standard ML with subtyping and inheritance [MMM91].

*Supported by the Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-90-J-1110

We distinguish between three levels of "entities" in our language: *specifications* at the highest level, *types* and *structures* in the middle, and *values* at the bottom.[1] Specifications consist of *interface* and *constraint* specifications. Specifications are inhabited by structures, which consist of zero, one, or more types and operations on those types. A type, in turn, denotes a set of values. Types, in some sense, pre-exist values: Every value has a type and comes into existence if and when its type is defined, typically as part of a structure definition. This may be at language design time (for built-in types) or at program design time (for user-defined types). A specification corresponds to its use in algebraic specification [EM85]; in object-oriented terms it can be roughly equated to the (public) interface of a class along with its invariants and assertions (see, e.g., Eiffel [Mey88]). In algebraic terms a structure is a valid model of a specification, and the types in a structure correspond to the carrier sets of such a model; in object-oriented terms the types in a structure represent the data part of a class implementation, and the operations stand for the methods. Detailed comparisons with other languages are given in Section 2.

We distinguish between the *explicit* language and the *implicit* language. Static and dynamic semantics are specified only for the explicit language (c.f. [MH88]). The programmer may, however, write in an abbreviated style relying on the programming environment to fill in missing information, interacting with it, if necessary. Thus the task of completing an implicit program can be seen as a purely combinatorial task that does not have to be supported in a uniform fashion in all programming environments since it is outside the language definition proper. Several pieces of information may be inferred by a programming environment:

- "classical" implicit type information, which models universal polymorphism;,
- applications of conversion functions, which models subtyping (c.f. [BCGS89]);
- structure resolving, which models static and dynamic overload resolution.

Whereas the type-theoretic ideas in our approach have appeared in various forms before, their combination and application to object-oriented programming appear to be new.

Instead of implicit subtyping we may define explicit conversion functions between types; for example, we may define a conversion from integers to reals and vice versa, or a conversion from cartesian colored points to cartesian points. This allows subtyping even for types defined as part of structures that do not satisfy the rule of [CCH+89], which has been criticized by Meyer as too restrictive [Mey89] and has been adopted in an even more restrictive fashion in [MMM91].

As pointed out before, a class in the object-oriented sense is modeled by a specification for a single type and, by extension, all structures that satisfy it, either by declaration or implicitly. The type of a "generic" object of such a class is represented by the existential type $\exists s : SPEC.\,|s|$ where $|s|$ denotes the type in structure s. An element of this type has two components, a structure s that satisfies specification $SPEC$ and a value of type $|s|$. This is an important difference from the approach taken in XML+ [MMM91]: there an object has also two components, a type component and a value component. The value component consists of all data *and* all their methods. So for two XML+ objects we cannot be sure whether their methods or their data representations are identical. In particular, binary methods that operate on objects with identical representation type are problematic in that approach.

Dynamic binding can be achieved by using the structure component of a generic object and selecting from it the corresponding operation and applying it to the value part of an object. For example, $x.f(w)$ in a language such as Eiffel is translated into

[1] Actually there is an even higher level, *modules*, that contain bindings for entities of all lower levels.

$$\text{let } < s : SPEC, v : |s| >= x \text{ in } s.f(v, w)$$

or

$$|x| .f(val(x), w)$$

where x is a generic object consisting of a structure s and a value v. In the implicit language we could have written $f(x, w)$ with the idea that the conversion *val* and the structure modifier $|x|$ can be inferred. Inferring structure modifiers of this sort is generally *dynamic overload resolution*; if the structure modifier is a (compile time) constant structure then it is essentially *static overload resolution*.

Specifications and structures are separate, as in XML+. This permits specifications with no structures or more than one structure, obviating the need for deferred classes and artificial subclasses. Also, specifications and structures can be independently reused — "inherited" in object-oriented lingo. This permits an independent development of specification enhancement and implementation specialization (c.f., [Sny87]).

Functional abstraction over structures of a specification makes it possible to treat members of an object class as first-class functions. This is possible because class members in our language do not have implicit arguments and because the range of applicability can be precisely described by a specification. For example, the method "translate" of object class Point has type

$$\text{translate}[s : Point](p : |s|, x, y : real) : |s|$$

The paper is organized as follows: Section 2 gives an overview of problems occurring in existing object-oriented languages, the functional language Haskell, and XML+. Section 3 introduces the key features of our language, called G in the remaining text. Section 4 describes the type-theoretic foundations of G along the lines of XML [MH88] and XML+ [MMM91]. We conclude with a summary of the contributions made and an outlook on further research. Appendix A contains a collection of program examples in G.

2 Some Problems of Other Approaches

2.1 Object-oriented Languages

Most strongly-typed object-oriented languages [Mey88,Str86] identify inheritance with subtyping, where the subtyping is based on extensible record types partially ordered with respect to a subtyping relation. This view appears too restrictive when it comes to modeling certain algebraic structures. Common properties of classes are typically factored out in a common superclass, so that heterogeneous structures can be constructed. To illustrate this, let us consider the following example. We define a class of partially ordered objects with the following signature:

```
class PartialOrder is
        less: PartialOrder -> Bool
```

Now we refine `PartialOrder` to a class `Int` with an addition operation, as in

```
class Int superclass PartialOrder is
        less: Int -> Bool
```

However, the definition of method `less` in class `Int` violates the contravariance rule for function subtyping. Hence it is not possible to define `Int` as a subtype of `PartialOrder`. This example shows

how the requirement that inherited classes be subtypes of their superclasses guarantees type safety but inhibits flexibility.

Trading off in the other direction, Eiffel abandons the contravariance rule in its type system and gives up static type safety for the sake of greater flexibility. Therefore we can actually create a class `PartialOrder` of partially ordered objects and several subclasses of `PartialOrder`, e. g. `Int` and `String`. Indeed, the Eiffel model allows us to write the following code:

```
bool compare(PartialOrder x, y)
        return x.less(y);
n = Int(3);
s = String("three");
n.compare(s)
```

which would lead to a runtime type error.

Another weakness of object-oriented languages is the identification of specification and implementation in a single construct. (An exception is the language Emerald [BHJL86], which separates specification and implementation, but lacks a reuse mechanism.) While the specification and the implementation of a class may be syntactically separate in order to support modular coding, it is not possible to identify a single specification with *multiple implementations*. If we tried modeling the specification as a common superclass and the implementations as subclasses, we would run into two problems. The first one is the contravariance problem mentioned above. The second problem has to do with the *interoperability* of objects of the same specification, but different implementation. As an example, consider the problem of providing a hash table and a linked list implementation of a set. The problem appears in coding binary operations such as "union" so that they work on a pair of hash tables or a pair of linked list representations, but not necessarily on mixed pairs. In an object-oriented language union will always have to work on mixed pairs as well.

2.2 Haskell Type Classes

Haskell ([HW90]) provides an overloading mechanism ([WB89]) which is different from the subtyping found in traditional object-oriented languages and solves the problem described above. Instead of capturing common properties of several classes by deriving them from a common superclass, in Haskell such classes are explicitly declared to be instances of the same *type class*. Unlike a class in object-oriented languages, a type class is not a type itself. Instead, it specifies certain properties required of its instance types.

In Haskell, we could express the orderedness of type `Int` by defining `PartialOrder` as a type class, and declaring `Int` and possibly other types as instances of `PartialOrder`. We are allowed to construct hierarchies of type classes. We may define e. g. a type class `Num` with numerical operations as a subclass of `PartialOrder`. `Int` could then be made an instance of `Num`, along with other types such as `Float`.

Nevertheless, the Haskell model has several major shortcomings. First, Haskell does not provide a mechanism for inheritance at the implementation level. Each instance of a type class has to be implemented without reusing other implementations. Second, Haskell's type classes cannot be parametrized nor used as parameters of type constructors. For example, it is not possible to define a list over the type class `Num` whose members could be of any type declared as an instance of `Num`. Neither is it possible to define a type class `Set` parametrized by the element type. See [Ode90,OL91] for a detailed treatment.

2.3 XML+

Various disadvantages of existing object-oriented languages have been recognized and described in [Mit90b,MMM91]. XML+ is an extension of the Standard ML module system and improves over both the object-oriented and the Haskell style in a number of respects. A major shortcoming of object-oriented languages is the merging of specification and implementation of classes. One consequence of this merging is the difficulty of providing multiple implementations of the same class specification. Another problem stems from the observation that specification and implementation inheritance (extension) often work most naturally in opposite ways. Consider e. g. queues and stacks as seen in section A. It is natural to specify a queue as an extension of the specification of a stack, since a queue provides at least all the operations that are part of a stack. On the other hand, a stack implementation can easily be obtained from a queue implementation by hiding the operations that are not needed. However, we lose this abstraction if we implement a queue in terms of a stack, because the queue implementation would have to know the representation of the stack to provide the additional operations [Sny87].

XML+ separates specification and implementation, using an extension of ML signatures for specifications, and an extension of ML structures for implementations, combined with separate inheritance mechanisms. Separate mechanisms for specification and structure subtyping are provided. F-bounded polymorphism ([CHC90]) is used to allow polymorphism over families of structurally similar types of objects that do not necessarily have a subtyping relationship. Furthermore, XML+ structures support traditional *abstract data types*, i. e. pairs consisting of a representation type and a set of operations on that type, which are not present in existing object-oriented languages. XML+ introduces *internal interfaces* which are used by multiple implementations to interact with one another.

We have found some short-comings in the XML+ support for object-oriented programming, which we will illustrate below. First, we find it important to be able to specify that two distinct objects have identical representation type and methods. In existing object-oriented languages such as C++ it is normally the case that two instances of the same class differ semantically only in their state. XML+, however, lacks a mechanism to guarantee this, as illustrated by an example:

```
specification OBJ = spec
                      val get: unit -> int;
                      val put: int -> unit
                    end
structure Obj1: OBJ = struct
                        val i: ref int = ref 0;
                        fun get () = !i;
                        fun put k = i := k
                      end
structure Obj2: OBJ = struct
                        val i: ref int = ref 2;
                        fun get () = 2 * !i;
                        fun put k = i := k + 3
                      end
```

Given specification `OBJ`, we can declare objects that are instances of `OBJ` and therefore satisfy the signature requirements. However, we are free to implement the components of the instances in different ways as long as the signature is correct. There is no direct mechanism to construct several instances of the same implementation; it can be approximated using code reuse at structure level.

G provides specifications, structures, and objects. While specifications contain signature information, structures implement representation and operations of a class or an ADT. By creating

several instances of the same structure, we are sure that they are identical and only differ in their state.

The second problem stems from the inheritance mechanism for structures and appears in a number of object-oriented languages. Inheritance in XML+ can be described as a textual copying combined with a renaming and visibility control mechanism. Consider the following example:

```
structure Amount = struct
                     type t = int * int;
                     val v: t = (12, 95)
                   end
structure UseAmount = struct
                        copy Amount;
                        fun dollars () = #1 v;
                        fun pennies () = #2 v
                      end
```

This textual copy mechanism is not safe; it may lead to problems known from other languages such as Smalltalk [GR83]. The problem generally appears in the part of the code that is copying from the other structure. If `Amount` is redefined with the representation given below, a type error occurs in `UseAmount`.

```
structure Amount = struct
                     type t = int;
                     val v = 1295
                   end
```

At specification level, besides textual copying, XML+ provides extension and restriction of specifications, which result in subtypes and supertypes of the original specification, respectively. Unfortunately, it is not clear how a recursive specification is extended. By a recursive specification we mean one whose name appears in the signature of one of its components. Suppose we extend the specification

```
specification StackClass[type t] = spec
                                     fun push: t -> StackClass[t]
```

to another specification

```
specification QueueClass[type t] = extend StackClass[t] with
                                     fun enqueue: t -> QueueClass[t]
```

Does the component `push` of `QueueClass` have type `t -> StackClass[t]` or `t -> QueueClass[t]`? Clearly, the latter would be more desirable as we might want to push an element onto our queue first, and then enqueue another one; hence we want `push` to return a queue. Traditional object-oriented languages provide constructs such as `mytype` or `like current`, while G solves this problem by identifying the object type with the representation type, and not with the type of the whole structure.

Although XML+ contributes to the solution of various problems present in the object-oriented paradigm, it does not completely resolve certain issues regarding code reuse and class specification.

3 Specifications, Types, and Values

In this section, we will present the data abstraction mechanisms of our language G. Let us give an overview of the concepts and terms we use, before we address the technical issues.

3.1 Basic Concepts and Terminology

type: A type denotes a collection of (first-class) values.[2] A type is not useful by itself; it is normally defined as a component of a structure, which provides operations involving that type. Types are either primitive, such as `int`, `bool`, `string`, or constructed from primitive types, such as function types, ML-style datatypes, etc. In G, monomorphic and polymorphic functions are considered first-class.

value: A value is an inhabitant of a type. Values are exactly the first-class entities in G, i. e. they can be returned as the result of a conditional expression, passed as a function parameter, or stored in a variable. Examples are `3, false, fn[t :: Type] (x : t) => x`.

structure: Structures in G are viewed the same way as in ML. A structure is an encapsulation unit that consists of zero, one, or more types, and zero, one, or more values. Structures inhabit specifications; although we could say that specifications are types of structures, we prefer to use the term "type" exclusively for types of first-class values. Structures containing one or more types are not first-class entities, since that would inhibit static typing. They are at the same level as types; in fact, a type may be identified with a structure containing only that (representation) type, but no operations. All predefined types are actually representation types of structures that define the operations on them.

functor: A functor is a structure template parametrized by (compile-time) values, types, or structures. Functors are inhabitants of parametrized specifications.

specification: A specification defines the interface of a structure, i. e. its visible components.[3] A structure is said to inhabit, or *implement* a specification if it provides definitions for all the entities required by the specification. Specifications can be parametrized by values, types, or structures; in that case, they are specifications of functors. So far, our specifications correspond to ML signatures or Ada package specifications. Furthermore, specifications can be parametrized with respect to other signatures.

module: A module is a compilation or library unit that contains bindings for entities of all lower levels, i. e. specifications, structures, types, and values.

hidden type and generic object: In many situations, we want to have objects that are instances of some implementation of a specification, but we do not care which one. Such *generic* objects consist of a (hidden) structure component and the object value, whose type is the representation type of the structure component.[4] They provide dynamic dispatching on subclasses in the sense that the structure component is hidden and may be locally opened in order to access the methods implemented on that particular representation type. The value component can be accessed only within an open statement; this restriction allows us to treat hidden types as first-class types.

reuse: Code reuse, or inheritance, can occur both at structure level and specification level. Reuse at structure level is not yet well-understood (see the discussion in [Mit90b]), and is currently handled by textual copying, although we are not satisfied with this method. At specification

[2] Hence in type-theoretic terms, types in G are "small" types. See Section 4 for a brief discussion of polymorphic functions.

[3] Optionally a specification can also contain constraints, which are, however, of no further relevance here.

[4] Such hidden types are called existential types in type-theoretic terminology, see [MP88]

level, new specifications can be derived from previous ones by copying, adding, or omitting parts of the specification.

subtyping and conformity: At type level, G provides no implicit subtyping; instead, explicit conversion functions may be used. This gives us higher flexibility for forms of subtyping that do not satisfy the restrictive contravariance rule for record subtyping [CHC90]. On the other hand, we have implicit conformity between specifications; for example, when a parameter is specified by a required interface, any structure that satisfies that interface may be passed.

3.2 Specification and Implementation

Let us now demonstrate how types can be specified and implemented. A specification states the abstract properties of a type, i. e. how a type will be used, but does not specify any concrete implementation details (except possibly for representation-independent bodies of operations, as we will see shortly). Consider the following example of a parametrized stack:

```
spec Stack[elem :: Type] = stack :: Type with
        new: stack
        push: stack * elem -> stack
        pop: stack -> stack
        top: stack -> elem
        isempty: stack -> bool
        -- constraints
```

Note that `Type` itself is a specification. Indeed, it is the most general specification in the sense that it does not specify any operations on its instance types.

Structures are instances of specifications and describe how abstract objects are implemented. The following structures give two alternative implementations of the specification `Stack`:

```
struct liststack[e :: Type] :: Stack[e] = List[e] with
        new = nil
        push(l, e) = e :: l
        pop = tl
        top = hd
        isempty(l) = (l = nil)

struct arraystack[e] = { a: Array[0..MAXSIZE] of e, i: 0..MAXSIZE } with
        new = { a = new[Array[0..MAXSIZE]], i = 0 }
        push(s, e) = s{a := s.a[i+1 := e], i := s.i + 1}
        pop(s, e) = s{i := s.i - 1}
        top(s) = s.a[s.i]
        isempty(s) = (s.i = 0)
```

They are functional implementations in the sense that they do not carry a state. A more "object-oriented," imperative implementation could be given by

```
struct refliststack[e] :: Stack[e] = ref List[e] with
        new = ref nil
        push(s, e) = (s := x :: !s; s)
        pop(s) = (s := tl(!s); s)
        top(s) = hd(!s)
        isempty(s) = (!s = nil)
```

Let us now demonstrate how specifications can be reused (inherited). From an abstract point of view, a queue is a stack with some more operations. When specifying a queue, it is natural to use the specification of `Stack` and tack on the additional operations.

```
spec Queue[elem:: Type] = Stack[elem] with stack as queue and
        append: elem * queue -> queue
        delete: queue -> queue
        last: queue -> elem
        isfull: queue -> bool
```

Note how the virtual representation `stack` is renamed to `queue` and used in the signature of the additional operations. The representation corresponds to "mytype" in typical object-oriented languages.

Having just seen reuse of specifications, let us introduce reuse of implementations, for which G provides a separate, quite flexible mechanism. Aiming at implementing queues, we start by extending the list implementation of a stack, and then giving an array-based implementation of a queue not inherited from any other structure.

```
struct listqueue[elem] :: Queue[elem] = liststack[elem] with
        (liststack[elem]): listqueue[elem] -> liststack[elem]
                        -- automatically inferred type conversion
        append(e, (c :: q)) = c :: append(e, q)
        append(e, nil) = [nil]
        delete(c :: nil) = nil
        delete(c :: l) = c :: delete(l)
        last(c :: nil) = c
        last(c :: l) = last(l)
        isfull(l) = false

struct arrayqueue[elem] :: Queue[elem] = { a: Array[0..MAXSIZE] of elem,
                i, j: 0..MAXSIZE, full: bool} with
        new = { a = new[Array[0..MAXSIZE], i = MAXNUM, j = 0, full = false }
        push(q, e) = if isfull(q) then raise full else
                q{a := q.a[j := e], j := next(q.j), full =
                        (prev(q.j) = q.i)}
        pop(q) = if isempty(q) then raise empty else
                q{j := prev(q.j)}
        top(q) = if isempty(q) then raise empty else
                q.a[prev(q.j)]
        append(e, q) = if isfull(q) then raise full else
                q{a := q.a[i := e], i := prev(q.i), full =
                        (prev(q.j) = q.i)}
        delete(q) = if isempty(q) then raise empty else
                q{i := next(q.i)}
        last(q) = if isempty(q) then raise empty else
                q.a[next(q.i)]
        isempty(q) = not isfull(q) and (prev(q.j) = q.i)
        isfull(q) = q.full
        -- auxiliary definitions
        next(i: int) = if i = MAXSIZE then 0 else i+1
        prev(i: int) = if i = 0 then MAX
```

Having written such an implementation of a queue, one might be curious whether it can be made into a stack implementation simply by getting rid of part of the operations provided. By giving

`arrayqueue` the signature `Stack`, we hide the operations in `arrayqueue` that are not part of `Stack`. As elaborated in section 2, existing object-oriented languages do not typically provide this flexibility:

```
struct arraystack2[elem] :: Stack[elem] = arrayqueue[elem]
```

3.3 Hidden Types and Dynamic Dispatching

Suppose we are given a specification `Point` along with some implementations (which we are not showing).

```
spec Point = t :: Type with
        new: real * real -> t
        x: t -> real
        y: t -> real
        r: t -> real = sqrt(x(p)^2 + y(p)^2)
        theta: t -> real = arctan(y(p)/x(p))
        eq(p: t, q: t): bool = x(p) = x(q) and y(p) = y(q)
        translate(p: t, u: real, v: real): t =
                new(x(p)+u, y(p)+v)
        scale(p: t, s: real): t =
                new(x(p)*s, y(p)*s)
        sqdistance(p: t, q: t): real =
                (x(p) - x(q))^2 + (y(p) - y(q))^2
```

A generic point would then have the type

```
type AnyPoint = some P :: Point with |P|
```

Another example of dynamic dispatching, which also involves multiple inheritance, is given in Appendix A.

4 Type-theoretic Aspects

The G type system is based on an explicitly typed, predicative lambda calculus and can be considered as an extension of the type systems described in [Mac86,MH88,MMM91]. In this section, we will briefly review the concepts underlying those type systems and then outline the G type system.

XML is a language introduced in [MH88] in order to explain more precisely the type system of Standard ML, including its module facility. The type system of XML provides two universes of types, U_1 and U_2, which are informally called "small" and "large" types, respectively. This separation reflects the phase distinction between the static evaluation of modules and the dynamic evaluation of values in Standard ML. The universe of all values, U_0, contains all entities whose types are members of U_1. XML may be defined relative to an arbitrary collection of base types, e. g. integers and booleans, and user-defined algebraic free types. Its small types include any type expression constructed using only base types, monomorphic type variables, and the function space constructor $\rightarrow$.

The large type universe of XML corresponds to Standard ML's polymorphic types and module facility. U_2 contains U_1 itself, polymorphic functions, and types constructed from other members of U_2 using general sum and product operations. We will now review these operations as introduced in [Mac86].

Let A be a set, and B a family of sets indexed by A, meaning that $B(a)$ is again a set for each $a \in A$. Then the *general product* of A and B, written $\Pi x : A.B(x)$ is the set of functions f from A to the union $\cup_{x \in A} B(x)$ such that for each $a \in A$, we have $f(a) \in B(a)$, i. e.

$$\Pi x : A.B(x) = \{f \in A \rightarrow \cup_{x \in A} B(x) \mid \forall a \in A.f(a) \in B(a)\}$$

Note that in the degenerate case in which B is constant, the general product reduces to the function space $A \rightarrow B$.

The *general sum*, written $\Sigma x : A.B(x)$, for a set A and a family of sets B indexed by A, is the set of pairs $< a, b >$ such that $a \in A$ and $b \in B(a)$, i. e.

$$\Sigma x : A.B(x) = \{< a, b > \in A \times \cup_{x \in A} B(x) \mid b \in B(a)\}$$

In the degenerate case, the general sum reduces to the cartesian product $A \times B$. We may apply projection functions $|.|$ (witness) and *val* to members of general sum types which return the first and second component of the member, respectively.

Viewing types as sets, XML incorporates general sum and product types into U_2 by requiring that the index type A and the types contained in the family B be members of U_2. The following are examples of general sum and product types and members of such types:

$$< int, 3 >: \quad \Sigma t : U_1.t$$
$$nil : \quad \Pi t : U_1.list(t)$$

As seen in [Mac86,MH88], the signatures of Standard ML may be viewed as syntactic sugar for general sum types, and ML structures and functors as a notation for members of general sum and product types, respectively. Although functor signatures are not provided in ML, they could be described by general product types.

The language XML+ is presented in [MMM91] and can be seen as a generalization of XML incorporating several new features. It generalizes signatures by providing not only structure signatures in form of general sums, but also functor signatures in form of general product types. In addition, XML+ features parametrized signatures, which have types of the form $U_2 \rightarrow U_2 \rightarrow \ldots \rightarrow U_2$, technically leaving the boundaries of U_2. Another feature included in XML+ is an impredicative treatment of existential types and a related implicit coercion from U_2 to U_1. We will give a brief overview of existential types and their use in XML+.

Existential types are introduced in [MP88,CW85] and model the programming language concept of type abstraction. Using the notation of [Mac86], an existential type is expressed as $\exists t : U_1.B(t)$, where B is a type expression possibly containing free occurrences of t. Values of such types are created by expressions of the form $\mathbf{hide}_{\exists t:U_1.B(t)} \tau M$, where τ is a U_1 type and M is an expression of type $B(t)$. The only expression available on members of existential types has the form **open** M **as** $x[t]$ **in** N. It has type ρ, assuming $M : \exists t : U_1.B(t)$ and $x : B(t) \Rightarrow N : \rho$, with the restriction that t does not leave the scope of N, i. e. appear free in ρ or the type of any variable appearing free in N. Locally within N, t refers to the type component, and x to the value component. Although existential types, similarly to general sum types, have a type component which is a member of U_1, they can be considered as members of U_1. This is possible because the type component is hidden and may be accessed only locally as an opaque type newly generated with each open operation.

XML+ provides signatures as general sum types; however, the structures described by those signatures may be treated as members of existential types so that they have small types. This flexibility is achieved by recognizing that there is a canonical coercion function

$$hide : \Sigma t : U_1.B(t) \rightarrow \exists t : U_1.B(t)$$

which simply hides the identity of the type component of a structure and is reflected in the typing rules.

G differs from XML in several ways. While maintaining a hierarchy of two type universes, it combines general sum types in U_2 with a modified notion of impredicative polymorphism and bounded existential types. Within the universe U_1, G identifies structures with the underlying representation types. We will now elaborate on these characteristics.

Polymorphism in G appears at two levels. Since it is desirable to treat polymorphic functions as first-class values, we consider polymorphic types such as $\Pi t : U_1.B(t)$, or commonly written $\forall t : U_1.B(t)$, as types in U_1. We can now write first-class functions parametrized by polymorphic functions, e. g.

$$g = \lambda f : (\forall t : U_1.t \rightarrow t).(f\ \mathit{int}\ 3, f\ \mathit{bool}\ \mathit{false})$$

This option is described in [MH88], and the question whether the extension of XML with this sort of impredicative polymorphism is strongly normalizing still appears to be open. However, it is believed to be strongly normalizing [Mit90a], since the code of polymorphic functions is completely oblivious to the type parameter. Analogously, we view functors of the form $\forall s : U_2.B(s)$, where $B(s) : U_2$, as members of U_2, thereby allowing us to parametrize functors by other functors.

In addition to existential types such as $\exists t : U_1.B(t)$, G provides signature-bounded existential types of the form $\exists t : C.B(|t|)$ as U_1 types, where $C : \Sigma t : U_1.F(t)$ and $|t|$ denotes the first (witness) component of t (see [CW85,Mac86]). Such types are used to model partial abstraction, meaning that the only known property of the hidden structure is its membership in a signature C. They are useful when modeling object-oriented programming, where we have objects with a known interface but a hidden representation. As an example, consider

$$\begin{aligned} C &= \Sigma t : U_1.t \rightarrow \mathit{bool} \\ S &= < \mathit{int}, \lambda x : \mathit{int}.x \leq 0 >: C \\ s &= \mathbf{hide}_{\exists t:C.|t|} S\ 3 \end{aligned}$$

Now we can use the square operation when we locally open s as in

$$\mathbf{open}\ s\ \mathbf{as}\ x[t]\ \mathbf{in}\ (\mathit{val}(t))(x) : \mathit{bool}$$

In contrast to XML+, which models classes as general sum types and objects as their members, G has a 3-level hierarchy viewing objects as values, object types (classes) as U_1 types, and signatures (class interfaces) as U_2 types. At the level of U_1, we identify structures that group a small type with operations on that type with the witness type itself, which we view as a mere set of values. As seen in the previous example, C is a class interface specifying a square operation, S is a structure with interface C, and 3 is an object of class S, i. e. a value of type $|S|$.

G has a notion of signature conformity similar to XML's.

5 Conclusion

We have developed a language framework that integrates algebraic, functional, and object-oriented programming in a uniform way. We have shown that abstraction over structures plays a critical role in offering flexible manipulation of both homogeneous and heterogeneous data. This combines the advantages of algebraic and object-oriented programming.

Several important open problems remain. Type reuse and type derivation need to be worked out carefully to overcome the non-robust nature of literally copying structures. Furthermore, the generalized type inference problem of completing programs in our implicit language need to be addressed where types, structure modifiers and conversion functions may be elided.

Acknowledgments

We would like to thank all members of the Griffin group for extensive and lively discussions providing valuable stimulation and feedback; in particular, Malcolm Harrison for advocating a style of object-oriented programming based on algebraic data types rather than on pure objects; Chih-Hung Hsieh for proposing the use of typesets as a means of describing collections of types; and Edmond Schonberg for detailed comments on the paper. We would further like to thank Martin Odersky for helpful suggestions and for sharing his insights on locally polymorphic types with us.

References

[BCGS89] V. Breazu, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Proc. Logic in Computer Science (LICS)*, pages 112–129, 1989.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 78–86, October 1986.

[CCH+89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Functional Programming and Computer Architecture*, pages 273–280, 1989.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 125–135, Jan. 1990.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[D+90] R. Dewar et al. Reference manual for the Griffin prototyping language. In progress, December 1990.

[EM85] H. Ehrig and B. Mahr. *Algebraic Specification*, volume 1/2 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[HW90] P. Hudak and P. Wadler. *Report on the Programming Language Haskell*, April 1990. (editors).

[Mac86] D. MacQueen. Using Dependent Types to Express Modular Structure. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 277–286. ACM, Jan. 1986.

[Mey88] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[Mey89] B. Meyer. Static typing for eiffel. posted to comp.lang.eiffel, July 1989.

[MH88] J. Mitchell and R. Harper. The essence of ML. In *Proc. Symp. on Principles of Programming Languages*. ACM, Jan. 1988.

[Mit90a] J. Mitchell. Private communication, Nov. 1990.

[Mit90b] J. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 109–124, 1990.

[MMM91] J. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1991.

[MP88] J. Mitchell and G. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.

[MTH90] R. Milner, M. Tofte., and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Ode90] M. Odersky. Locally polymorphic types. submitted to SIGPLAN '91, Nov. 1990.

[OL91] M. Odersky and K. Läufer. Type classes are signatures of abstract types. Submitted to FPCA '91, March 1991.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[Sny87] A. Snyder. Inheritance and the development of encapsulated software components. In *Research Directions in Ojbect-Oriented Programming*, pages 165–188. MIT Press, 1987.

[Str86] B. Stroustrop. *The C++ Programming Language*. Addison-Wesley, 1986.

[Uni83] United States Department of Defense. *Reference Manual for the ADA Programming Language*. Springer-Verlag, 1983.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.

A A Collection of Examples in G

This section contains a number of examples that we considered during the design of our language.

A.1 Points, Circles, and Rectangles: Hidden Types and Dynamic Dispatching

We will start with another example of dynamic dispatching, which also involves multiple inheritance. It is based on the example in Section 3. Given the specification for `Point` from Section 3, we add two specifications, one for graphical objects and one for colored objects.

```
type Color = { red, green, blue }

spec ColorObj = colorobj :: Type with
        color: colorobj -> Color

spec GraphObj = graphobj :: Type with
        draw: graphobj -> void
        scale: graphobj * real -> graphobj
```

Using multiple inheritance, we obtain a new specification for graphical objects that also have a color:

```
spec ColGraphObj = ColorObj with colorobj as colgraphobj and
        GraphObj with graphobj as colgraphobj
```

Let us define some geometric objects we can draw. Note how they use the type `AnyPoint` we defined above, since we do not care how the points constituting the rectangle are represented.

```
spec CircleObj = GraphObj with graphobj as circle and
        new: AnyPoint * real -> circle
        center: circle -> AnyPoint
        radius: circle -> real
        scale: circle * real -> circle

spec RectangleObj = GraphObj with graphobj as rectangle and
        new: AnyPoint * AnyPoint -> rectangle
        lowerleft: rectangle -> AnyPoint
        upperright: rectangle -> AnyPoint
```

Here are implementations for Circle and Rectangle.

```
struct gencircle :: CircleObj = { center: AnyPoint, radius: real } with
        new(p, r) = { center = p, radius = r }
        center(c) = c.center
        radius(c) = c.radius
        scale(c, s) = c{ radius = c.radius * s }
        draw(c) = let <s, v> = c.center in
                  drawcircle(s.x(v), s.y(v), p.radius)

struct genrectangle :: RectangleObj =
                { lowerleft: AnyPoint, upperright: AnyPoint } with
        new(p, p') = { lowerleft = p, upperright = p' }
        lowerleft(r) = r.lowerleft
        upperright(r) = r.upperright
        scale(r, s) = let <t, v> = r.upperright in
                        r{ r.upperright = t.scale(v, s) }
        draw(r) = let <tl, vl> = r.lowerleft and
                      <tr, vr> = r.upperright in
                  drawrectangle(tl.x(vl), tl.y(vl), tr.x(vr), tr.y(vr))
```

Finally we come to the interesting part. We define a type for any graphical object and a function which locally dispatches such an object to its proper drawing function. Remember that such objects are actually pairs of a hidden type component, which gets bound to `s`, and a value of the representation type contained in the hidden type component, here bound to `v`.

```
type AnyGraphObj = some G: GraphObj with |G|

draw(g: AnyGraphObj) = let <s, v> = g in s.draw(v)
```

Now we can define a *heterogeneous* list type of graphical objects and draw such a list.

```
struct GraphList = List[AnyGraphObj] with
        draw: List[AnyGraphObj] -> void
        draw(nil) = ();
        draw(<s,v>::l) = s.draw(v); draw(l)
```

Hidden types actually offer more than what we just showed. The following example shows how they can be used to group two points of the same type, where we only care that the type is the same, but not which particular implementation of the specification `Point`.

```
type APPair = some P :: Point with |P| * |P|

eq(pp: APPair): bool = |pp|.eq(pp.1, pp.2)
```

Using such a pair type, we can define a safe equality function that takes a pair of points and dispatches them to the equality function implemented in the common point structure P. Generally, this makes binary operations on types dynamically dispatchable without endangering static type safety or requiring multiple argument dispatching.

A.2 Based Set: Objects and Abstraction

Based sets play an important role in the implementation of the language SETL [SDDS86]. At specification level, a based set is parametrized by the element type and provides a set type and a "translated" element type, both of which are abstract. Various set operations, overloaded for use with based or unbased sets, are provided.

```
spec basedSet[elem :: Eq] =
        type set
        type belem

        in: elem * set -> bool
        with: set * elem -> bool
        less: set * elem -> bool

        in: belem * set -> bool
        with: set * belem -> bool
        less: set * belem -> bool

        lookup: elem -> belem

        union: set * set -> set
        -- etc.
```

A simple, inefficient implementation of a based set can be given by representing the set as a bitvector and the translated elements as indices into the bitvector. In addition, we need a hidden variable representing the translation from indices to actual elements, implemented as a sequence[5] We give implementations of the different set operations.

```
struct basedSet[elem :: Eq] =
        type set = Seq[Bool]
        type belem = Nat
        type bset = Seq[elem]
        var base: bset

        in(b: belem, s: set): bool = if b > # base then false else s[belem]
        with(s: set, b: belem): set = s[b := true]
        less(s: set, b: belem): set = s[b := false]
```

[5]The efficiency could be improved by using an associative data structure, e. g. a hash table.

```
        lookup(e: elem): belem = lookup(e, base)-- only this lookup is exported
        lookup(e: elem, bs: bset): belem =      -- this one is hidden
                if # s = 0 then
                  0
                else
                  1 + lookup(e, s[2..]) -- this is the inefficient part

        in(e: elem, s: set): bool =
                s[lookup(e)] handle subscript_error => false
        with(s: set, e: elem) =
                let l = lookup(e) in
                  if l = 0 then
                        base := base ++ [e]
                        s ++ [true]
                  else
                        s[# l := true]  -- assuming that this maintains the
                                        -- sequence contiguous by inserting
                                        -- appropriate false values
        less(s: set, e: elem) =
                s[lookup(e) := false] handle subscript_error => s

        -- union etc. implemented bitwise, base need not be updated
```

Note that the hidden variable `base` would be called a class variable in object-oriented terminology. Given this implementation, we can create variables of type `set`; the base is updated as operations on the variables are performed.

A.3 A Package for Vectors and Matrices: Multiple Implementations and Hidden Types

This example illustrates the use of a structure implementing several related types. We show a specification requiring two types, `vec` and `mat`, and operations involving these types.

```
spec VecMat[elem :: Num] =
        vec :: Type
        mat :: Type

        newvec: int * int -> vec                -- constructors
        newmat: int * int * int * int -> mat

        +: vec * vec -> vec                     -- arithmetic operations
        -: vec -> vec
        -: vec * vec -> vec
        *: vec * vec -> elem
        *: mat * vec -> vec
        *: vec * mat -> vec
        *: mat * mat -> mat

        [.]: vec * int -> elem                  -- indexing operations
        [.]: mat * int -> vec
        [,.]: mat * int -> vec
        [.,.]: mat * int * int -> elem
```

```
        -- etc.
```

Matrices and vectors may be dense or sparse. It is useful to provide implementations which efficiently cover both cases:

```
struct denseVecMat[elem] :: VecMat[elem] =
        type vec = Array[elem]
        type mat = Array[Array[elem]]
        -- etc.

struct sparseVecMat[elem] :: VecMat[elem] =
        type sparseVecElem = elem * int
        type sparseMatElem = elem * int * int
        type vec = Seq[sparseVecElem]
        type mat = Seq[sparseMatElem]
        -- etc.
```

We can now define generic object types for systems and solutions of linear equations. The type `anyLinEq` defines a tuple containing a vector and a matrix instantiated from the same, arbitrary implementation. We can code functions independently of the implementation of the objects passed as parameters; the Gaussian elimination function below works on dense or sparse vectors and matrices.

```
type anyLinEq = some vm :: VecMat with vm.mat * vm.vec
type anyLinSol = some vm :: VecMat with Seq[vm.vec]

gauss(leq: anyLinEq): anyLinSol =
        open leq as vm[vec,mat] in
                -- some Gauss elimintation operations
                -- using operations defined in VecMat
                -- computing seqOfSpanningVecs
                hide vm seqOfSpanningVecs
```