Publication Series of the John von Neumann Institute for Computing (NIC)
NIC Series Volume 27

John von Neumann Institute for Computing (NIC)

Jörg Striegnitz and Kei Davis (Eds.)

Joint proceedings of the Workshops on

Multiparadigm Programming with Object-Oriented Languages (MPOOL'03)

Declarative Programming in the Context of Object-Oriented Languages (DP-COOL'03)

organized by

John von Neumann Institute for Computing

in cooperation with the

Los Alamos National Laboratory, New Mexico, USA

NIC Series Volume 27

ISBN 3-00-016005-1

Die Deutsche Bibliothek – CIP-Cataloguing-in-Publication-Data
A catalogue record for this publication is available from Die Deutsche Bibliothek.

Publisher: NIC-Directors
Distributor: NIC-Secretariat
Research Centre Jülich
52425 Jülich
Germany
Internet: www.fz-juelich.de/nic
Printer: Graphische Betriebe, Forschungszentrum Jülich

NIC Series Volume 27
ISBN 3-00-016005-1

Table of Contents

A Static C++ Object-Oriented Programming (SCOOP) Paradigm Mixing Benefits of Traditional OOP and Generic Programming

Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss

EPITA Research and Development Laboratory
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre, France
firstname.lastname@lrde.epita.fr

Abstract. Object-oriented and generic programming are both supported in C++. OOP provides high expressiveness whereas GP leads to more efficient programs by avoiding dynamic typing. This paper presents SCOOP, a new paradigm which enables both classical OO design and high performance in C++ by mixing OOP and GP. We show how classical and advanced OO features such as virtual methods, multiple inheritance, argument covariance, virtual types and multimethods can be implemented in a fully statically typed model, hence without run-time overhead.

1 Introduction

In the context of writing libraries dedicated to scientific numerical computing, expressiveness, reusability and efficiency are highly valuable. Algorithms are turned into software components that handle mathematical abstractions while these abstractions are mapped into types within programs.

The object-oriented programming (OOP) paradigm offers a solution to express reusable algorithms and abstractions through abstract data types and inheritance. However, as studied by Driesen and Hölzle [18], manipulating abstractions usually results in a run-time overhead. We cannot afford this loss of performance since efficiency is a crucial issue in scientific computing.

To both reach a high level of expressiveness and reusability in the design of object-oriented scientific libraries and keep an effective run-time efficiency for their routines, we have to overcome the problem of “abstractions being inefficient”. To cope with that, one can imagine different strategies.

A first idea is to find an existing language that meets our requirements, i.e., a language able to handle abstractions within programs without any penalty at execution time. This language has to be either well-known or simple enough to ensure that a scientist will not be reluctant to use our library. Unfortunately we do not feel satisfied with existing languages; for instance LOOM and PolyTOIL by Bruce et al. [11, 9] have the precise flavor that we expect but, as prototypes, they do not feature all what a complete language can offer.

A second approach, chosen by Baumgartner and Russo [6] and Bracha et al. [8] respectively for C++ and Java, is to extend an existing expressive language by adding ad hoc features making programs more efficient at run-time. Yet, this approach requires a too great amount of work without any guarantee that extensions will be adopted by the language community and by compiler vendors. To overcome this problem, an alternate approach is to propose a front-end to translate an extended language, more expressive, into its corresponding primary language, efficient, such as Stroustrup [48] did with his erstwhile version of the C++ language. This approach has been made easier than in the past thanks to recently available tools dedicated to program translation, for instance Xt [56]. However, we have not chosen this way since we are not experimented enough with this field.

Another strategy is to provide a compiler that produces efficient source codes or binaries from programs written in an expressive language. For that, several solutions have been developed that belong to the fields of static analysis and partial evaluation, as described by Chambers et al. [14], Schultz [41], Veldhuizen and Lumsdaine [55]. In particular, how to avoid the overhead of polymorphic method calls is studied by Aigner and Hölzle [2], Bacon and Sweeney [4] for C++ and by Zendra et al. [57] for Eiffel. However, most of these solutions remain prototypes and are not implemented in well-spread compilers.

Last, we can take an existing object-oriented language and try to bend it to make some constructs more efficient. That was for instance the case of the expression templates construct defined by Veldhuizen [53] in C++, later brought to Ada by Duret-Lutz [19], and of mixin-based programming by Smaragdakis and Batory [43] in C++. These solutions belong to the field of the *generic programming* (GP) paradigm, as described by Jazayeri et al. [26]. This programming style aims at implementing algorithms as general so reusable as possible without sacrificing efficiency obtained by parameterization—related to the **`template`** keyword in C++ and to the `generic` keyword in Ada and Eiffel. However, from our experience in developing a scientific library, we notice several major drawbacks of GP that seriously reduce expressiveness and affect user-friendliness, whereas these drawbacks do not exist with "classical" OOP. A key point of this paper is that we do *not* subscribe to "traditional" GP because of these drawbacks. Said shortly, they have their origin in the unbounded structural typing of parameterization in C++ which prevents from having strongly typed signatures for functions or methods. Consequently, type checking at compile-time is awkward and overloading is extremely restricted. Justifications of our position and details about GP limitations are given later on in this paper.

Actually, we want to keep the best of both OOP and GP paradigms—inheritance, overloading, overriding, and efficiency—without resorting to a new language or new tools—translators, compilers, or optimizers. The advent of the C++ Standard Template Library, mostly inspired by the work of Stepanov et al. [46], is one the first serious well-known artifact of GP. Following that example a lot of scientific computing C++ libraries arose during the past few years(they are referenced by oonumerics [38]), one of the most predominant being Boost [7]. Meanwhile, due to the numerous features of C++, many related GP techniques appeared and are described in the books by Czarnecki and Eisenecker [17], Alexandrescu [3], Vandevoorde and Josuttis [52]. Moreover, Striegnitz and

Smith [47], Järvi and Powell [25], Smaragdakis and McNamara [44] have shown that some features offered by a non-object-oriented paradigm, namely the functional one, can be supported by the native C++ language. Knowing these C++ programming techniques, we then thought that this language was able to support an OOP-like paradigm without compromising efficiency. The present paper describes this paradigm, namely a proposal for "Static C++ Object-Oriented Programming": SCOOP.

This paper is composed of three parts. Section 2 discusses the OOP and GP paradigms, their limitations, existing solutions to overcome some of these limitations, and finally what we expect from SCOOP. Section 3 shows how SCOOP is implemented. Finally some technical details and extra features have been moved into appendices.

2 OOP, GP, and SCOOP

A scientific library offers data structures *and* algorithms. This procedural point of view is now consensual [34] although it seems to go against OOP. Actually, an algorithm is intrinsically a general entity since it deals with abstractions. To get the highest decoupling as possible between data and algorithms, a solution adopted by the C++ Standard Library and many others is to map algorithms into functions. At the same time, data structures are mapped into classes where most of the methods are nothing but the means to access data. Last, providing reusable algorithms is an important objective of libraries so we have to focus on algorithms. It is then easier to consider that algorithms and all other entities are functions (such as in functional languages) to discuss typing issues. For all these reasons, we therefore adopt in this section a function-oriented approach of algorithms.

2.1 About Polymorphisms

A function is polymorphic when its operands can have more than one type, either because there are several definitions of the function, or because its definition allows some freedom in the input types. The right function to call has to be chosen depending on the context. Cardelli and Wegner [13] outline four different kinds of polymorphism.

In **inclusion polymorphism**, a function can work on any type in a *type class*. Type classes are named sets of types that follow a uniform interface. Functional languages like Haskell allow programmers to define type classes explicitly, but this polymorphism is also at the heart of OO languages. In C++, inclusion polymorphism is achieved via two mechanisms: subclassing and overriding of virtual functions.

Subclassing is used to define sets of types. The **`class`** (or **`struct`**) keyword is used to define types that can be partially ordered through a hierarchy: i.e., an inclusion relation[1]. A function which expects a pointer or reference to a class `A` will accept an instance of `A` or any subclass of `A`. It can be noted that C++'s typing rules make no

[1] Inclusion polymorphism is usually based on a subtyping relation, but we do not enter the debate about "subclassing v. subtyping" [15].

difference between a pointer to an object whose type is exactly A and a pointer to an object whose type belongs to the type class of A[2].

Overriding of virtual functions allows types whose operations have different implementations to share the same interface. This way, an operation can be implemented differently in a subclass of A than it is in A. Inclusion polymorphism is sometime called *operation polymorphism* for this reason.

These two aspects of inclusion polymorphism are hardly dissociable: it would make no sense to support overriding of virtual functions without subclassing, and subclassing would be nearly useless if all subclasses had to share the same implementation.

In **parametric polymorphism**, the type of the function is represented using at least one generic type variable. Parametric polymorphism really corresponds to ML generic functions, which are compiled only once, even if they are used with different types. Cardelli and Wegner states that Ada's generic functions are not to be considered as parametric polymorphism because they have to be *instantiated explicitly* each time they are used with a different type. They see Ada's generic functions as a way to produce several monomorphic functions by macro expansion. It would therefore be legitimate to wonder whether C++'s function templates achieve parametric polymorphism. We claim it does, because unlike Ada's generics, C++'s templates are instantiated *implicitly*. In effect, it does not matter that C++ instantiates a function for each type while ML compiles only one function, because this is transparent to the user and can be regarded as an implementation detail[3].

These two kinds of polymorphism are called *universal*. A nice property is that they are open-ended: it is always possible to introduce new types and to use them with existing functions. Two other kinds of polymorphism do not share this property. Cardelli and Wegner call them *ad-hoc*.

Overloading corresponds to the case where several functions with different types have the same name.

Coercion polymorphism comes from implicit conversions of arguments. These conversions allow a monomorphic function to appear to be polymorphic.

All these polymorphisms coexist in C++, although we will discuss some notable incompatibilities in section 2.3. Furthermore, apart from virtual functions, the resolution of a polymorphic function call (i.e., choosing the right definition to use) is performed at compile-time.

2.2 About the Duality of OOP and GP

Duality of OOP and GP has been widely discussed since Meyer [32]. So we just recall here the aspects of this duality that are related to our problem.

Let us consider a simple function foo that has to run on different image types. In traditional OOP, the image abstraction is represented by an abstract class, Image, while

[2] In Ada, one can write access A or access A'Class to distinguish a pointer to an instance of A from a pointer to an instance of any subclass of A.

[3] This implementation detail has an advantage, though: it allows specialized instantiations (i.e., template specializations). To establish a rough parallel with *inclusion polymorphism*, template specializations are to templates what method overriding is to subclassing. They allow to change the implementation for some types.

a concrete image type (for instance `Image2D`) for a particular kind of 2D images, is a concrete subclass of the former. The same goes for the notion of "point" that gives rise to a similar family of classes: `Point`, which is abstract, and `Point2D`, a concrete subclass of `Point`. That leads to the following code[4]:

```
struct Image {
  virtual void set(const Point& p, int val) = 0;
};

struct Image2D : public Image {
  virtual void set(const Point& p, int val) { /* impl */ }
};

void foo(Image& input, const Point& p) {
  // does something like:
  input.set(p, 51);
}

int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

`foo` is a polymorphic function thanks to *inclusion through class inheritance*. The call `input.set(p, 51)` results in a run-time dispatch mechanism which binds this call to the proper implementation, namely `Image2D::set`. In the equivalent GP code, there is no need for inheritance.

```
struct Image2D {
  void set(const Point2D& p, int val) { /* impl */ }
};

template <class IMAGE, class POINT>
void foo(IMAGE& input, const POINT& p) {
  // does something like:
  input.set(p, 51);
}

int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

`foo` is now polymorphic through *parameterization*. At compile-time, a particular version of `foo` is instantiated, `foo<Image2D, Point2d>`, dedicated to the particular call to `foo` in `main`. The basic idea of GP is that all exact types are known at compile-time. Consequently, functions are specialized by the compiler; moreover, every function call can be inlined. This kind of programming thus leads to efficient executable codes.

[4] Please note that, for simplification purpose, we use **`struct`** instead of **`class`** and that we do not show the source code corresponding to the `Point` hierarchy.

The table below briefly compares different aspects of OOP and GP.

notion	OOP	GP
typing	named typing through class names so explicit in class definitions	structural so only described in documentation
abstraction (e.g., image)	abstract class (e.g., `Image`)	formal parameter (e.g., `IMAGE`)
inheritance	is the way to handle abstractions	is only a way to factorize code
method (set)	no-variant (`Image::set(Point, int)` `Image2D::set(Point, int)`)	— — —
algorithm (foo)	a single code at program-time (`foo`) and a unique version at compile-time (`foo`)	a single meta-code at program-time (`template<..> foo`) and several versions at compile-time (`foo<Image2D,Point2D>`, etc.)
efficiency	poor	high

From the C++ compiler typing point of view, our OOP code can be translated into:

$\text{type Image} = \{\ \text{set} : \text{Point} \rightarrow \text{Int} \rightarrow \text{Void}\ \}$

$\text{foo} : \text{Image} \rightarrow \text{Point} \rightarrow \text{Void}$

`foo` is restricted to objects whose types are respectively subclasses of `Image` and `Point`. For our GP code, things are very different. First, the image abstraction is not explicitly defined in code; it is thus unknown by the compiler. Second, both formal parameters of `foo` are anonymous. We then rename them respectively "I" and "P" in the lines below and we get:

$\forall\, \text{I},\quad \forall\, \text{P},\quad \text{foo} : \text{I} \rightarrow \text{P} \rightarrow \text{Void}$

Finally, if these two pieces of code seem at a first sight equivalent, they do not correspond to the same typing behavior of the C++ language. Thus, they are treated differently by the compiler and have different advantages and drawbacks. The programmer then faces the duality of OOP and GP and has to determinate which paradigm is best suited to her requirements.

During the last few years, the duality between OOP and GP has given rise to several studies.

Different authors have worked on the translation of some design patterns [22] into GP; let us mention Géraud et al. [23], Langer [27], Duret-Lutz et al. [20], Alexandrescu [3], Régis-Gianas and Poss [39].

Another example concerns the *virtual types* construct, which belongs to the OOP world even if very few OO languages feature it. This construct has been proposed as an extension of the Java language by Thorup [50] and a debate about the translation and equivalence of this construct in the GP world has followed [10, 51, 40].

Since the notion of virtual type is of high importance in the following of this paper, let us give a more elaborate version of our previous example. In an augmented C++ language, we would like to express that both families of image and point classes are related. To that aim, we could write:

```
struct Image {
  virtual typedef Point point_type = 0;
  virtual void set(const point_type& p, int val) = 0;
};

struct Image2D : public Image {
  virtual typedef Point2D point_type;
  virtual void set(const point_type& p, int val) { /* impl */ }
};
```

`point_type` is declared in the `Image` class to be an “abstract type alias” (**`virtual typedef`** `.. point_type = 0;`) with a constraint: in subclasses of `Image`, this type should be a subclass of `Point`. In the concrete class `Image2D`, the alias `point_type` is defined to be `Point2D`. Actually, the behavior of such a construct is similar to the one of virtual member functions: using `point_type` on an image object depends on the exact type of the object. A sample use is depicted hereafter:

```
Image* ima = new Image2D();
// ...
Point* p = new (ima->point_type)();
```

At run-time, the particular exact type of `p` is `Point2D` since the exact type of `ima` is `Image2D`.

An about equivalent GP code in also an augmented C++ is as follows:

```
struct Image2D {
  typedef Point2D point_type;
  void set(const point_type& p, int val) { /* impl */ }
};

template <class I>
where I {
  typedef point_type;
  void set(const point_type&, int);
}
void foo(I& input, const typename I::point_type& p) {
  // does something like:
  input.set(p, 51);
}

int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

Such as in the original GP code, inheritance is not used and typing is fully structural. On the other hand, a *where clause* has been inserted in `foo`’s signature to precise the nature of acceptable type values for `I`. This construct, which has its origin in CLU [29], can be found in Theta [28], and has also been proposed as an extension of the Java language [35]. From the compiler point of view, `foo`’s type is much more precise than in the traditional GP code. Finally, in both C++ OOP augmented with virtual types and C++ GP augmented with where clauses, we get stronger expressiveness.

2.3 OOP and GP Limitations in C++

Object-Oriented Programming relies principally on the inclusion polymorphism. Its main drawback lies in the indirections necessary to run-time resolution of virtual methods. This run-time penalty is undesirable in highly computational code; we measured that getting rid of virtual methods could speed up an algorithm by a factor of 3 [24].

This paradigm implies a loss of typing: as soon as an object is seen as one of its base classes, the compiler looses some information. This limits optimization opportunities for the compiler, but also type expressiveness for the developer. For instance, once the exact type of the object has been lost, type deduction (`T::deducted_type`) is not possible. This last point can be alleviated by the use of virtual types [51], which are not supported by C++.

The example of the previous section also expresses the need for covariance: `foo` calls the method `set` whose expected behavior is covariant. `foo` precisely calls `Image2D::set(Point2D,int)` in the GP version, whereas the call in the OOP version corresponds to `Image::set(Point,int)`.

Generic Programing on the other hand relies on parametric polymorphism and proscribes virtual functions, hence inclusion polymorphism. The key rule is that the exact type of each object has to be known at compile-time. This allows the compiler to perform many optimizations. We can distinguish three kinds of issues in this paradigm:

- the rejection of operations that cannot be typed statically,
- the closed world assumption,
- the lack of template constraints.

The first issues stem from the will to remain statically typed. Virtual functions are banished, and this is akin to rejecting inclusion polymorphism. Furthermore there is no way to declare an heterogeneous list and to update it at run-time, or, more precisely to dynamically replace an attribute by an object of a compatible subtype. These operations cannot be statically typed, there can be no way around this.

The closed world assumption refers to the fact that C++'s templates do not support separate compilation. Indeed, in a project that uses parametric polymorphism exclusively it prevents separate compilation, because the compiler must always know all type definitions. Such monolithic compilation leads to longer build times but gives the compiler more optimization opportunities. The C++ standard [1] supports separate compilation of templates via the **`export`** keyword, but this feature has not been implemented in mainstream C++ compilers yet.

The remaining issues come from bad interactions between parametric polymorphism and other polymorphisms in C++. For instance, because template arguments are unconstrained, one cannot easily overload function templates. Figure 1 illustrates this problem. When using inclusion polymorphism (left), the compiler knows how to resolve the overloading: if `arg` is an instance of a subclass of `A1`, resp. `A2`, it should be used with the first resp. second definition of `foo()`. We therefore have two implementations of `foo()` handling two different sets of types. These two sets are not closed (it is always possible to add new subclasses), but they are constrained. Arbitrary types cannot be added unless they are subtypes of `A1` or `A2`. This constraint, which distinguishes the two sets of types, allows the compiler to resolve the overloading.

In generic programming, such an overloading could not be achieved, because of the lack of constraints on template parameters. The middle column on Figure 1 shows a straightforward translation of the previous example into parametric polymorphism.

```
                        template<class A1>              template<class A1>
void foo(A1& arg)       void foo(A1& arg)               void foo(A1& arg)
{                       {                               {
  arg.m1()                arg.m1()                        arg.m1()
}                       }                               }

                        template<class A2>              template<>
void foo(A2& arg)       void foo(A2& arg) // illegal    void foo<A2>(A2& arg)
{                       {                               {
  arg.m2()                arg.m2()                        arg.m2()
}                       }                               }
```

Fig. 1. Overloading can be mixed with inclusion polymorphism (left), but will not work with unconstrained parametric polymorphism (middle and right).

Because template parameters cannot be constrained, the function's arguments have to be generalized *for any type* `A`, and *for any type* `B`. Of course, the resulting piece of code is not legal in C++ because both functions have the same type. A valid possibility (on the right of Figure 1), is to write a definition of `foo` for any type `A1`, and then *specialize* this definition for type `A2`. However, this specialization will only work for one type (`A2`), and would have to be repeated for each other type that must be handled this way.

Solving overloading is not the only reason to constrain template arguments, it can also help catching errors. Libraries like STL, which rely on generic programming, document the requirements that type arguments must satisfy. These constraints are gathered into *concepts* such as *forward iterator* or *associative container* [46]. However, these concepts appear only in the documentation, not in typing. Although some techniques have been devised and implemented in SGI's STL to check concepts at compile-time, the typing of the library still allows a function expecting a *forward iterator* to be instantiated with an *associative container*. Even if the compilation will fail, this technique will not prevent the compiler from instantiating the function, leading to cryptic error messages, because some function part of the *forward iterator* requirements will not be found in the passed associative container. Could the *forward iterator* have been expressed as a constraint on the argument type, the error would have been caught at the right time i.e. during the attempt to instantiate the function template, not after the instantiation.

2.4 Existing Clues

As just mentioned, some people have already devised ways to check constraints. Siek and Lumsdaine [42] and McNamara and Smaragdakis [31] present a technique to check template arguments. This technique relies on a short checking code inserted at the top of a function template. This code fails to compile if an argument does not satisfy its requirements and is turned into a no-op otherwise. This technique is an effective means of performing structural checks on template arguments to catch errors earlier. However, constraints are just *checked*, they are not *expressed* as part of function types. In particular, overloading issues discussed in the previous section are not solved. Overloading has to be solved by the compiler *before* template instantiation, so any technique that works after template instantiation does not help.

Ways to *express* constraints by subtyping exist in Eiffel [33] and has been proposed as a Java extension by Bracha et al. [8]. Figure 2 shows how a similar C++ extension could be applied to the example from Section 2.2.

```
concept image {
  typedef point_type;
  void set(const point_type& p, int val);
};

struct Image2D models image {
  typedef Point2D point_type;
  void set(const point_type& p, int val) { /* impl */ }
};

template <class I models image>
void foo(I& input, const typename I::point_type& p) {
  // does something with:
  input.set(p, 51);
}

int main() {
  Image2D ima; Point2D p;
  foo(ima, p);
}
```

Fig. 2. Extending C++ to support concept constraints

We have introduced an explicit construct through the keyword `concept` to express the definition of `image`, the structural type of images. This construct is also similar to the notion of signatures proposed by Baumgartner and Russo [6] as a C++ extension. Having explicitly a definition of `image` constraints the formal parameter `I` in `foo`'s type.

Some interesting constructions used to constrain parametric polymorphism or to emulate dynamic dispatch statically rely on a idiom known as the *Barton and Nackman trick* [5] also known as the *Curiously Recurring Template Pattern* [16]. The idea is that a super class is parameterized by its immediate subclass (Figure 3), so that it can define methods for this subclass.

For instance the Barton and Nackman trick has been used by Furnish [21] to constrain parametric polymorphism and simplify the Expression Template technique of Veldhuizen [53].

```
template <class T>
struct super
{
  void foo(const T& arg)
  {
    // ...
  }
};
```

```
struct infer : public super<infer>
{
  // ...
};
```

Fig. 3. The Barton and Nackman trick

2.5 Objectives of SCOOP

Our objective in this paper is to show how inclusion polymorphism can be almost completely emulated using parametric polymorphism in C++ while preserving most OOP features. Let us define our requirements.

Class Hierarchies. Developers should express (static) class hierarchies just like in the traditional (dynamic) C++ OOP paradigm. They can draw UML static diagrams to depict inheritance relationships between classes of their programs. When they have a class in OO, say `Bar`, its translation in SCOOP is a single class template: `Bar`[5].

Named Typing. When a scientific practitioner designs a software library, it is convenient to reproduce in programs the names of the different abstractions of the application domain. Following this idea, there is an effective benefit to make explicit the relationships between concrete classes and their corresponding abstractions to get a more readable class taxonomy. We thus prefer named typing over structural typing for SCOOP.

Multiple Inheritance. In the object model of C++, a class can inherits of several classes at the same time. There is no reason to give up this feature in SCOOP.

Overriding. With C++ inheritance come the notions of pure virtual functions, of virtual functions, and of overriding functions in subclasses. We want to reproduce their behavior in SCOOP but without their associated overhead.

Virtual Types. This convenient tool (see sections 2.2 and 2.3) allows to express that a class encloses polymorphic **`typedef`**s. Furthermore, it allows to get covariance for member functions. Even if virtual types does not exist in primary C++, we want to express them in SCOOP.

Method Covariance. It seems reasonable to support method covariance in SCOOP, and particularly binary methods. Since our context is static typing with parametric polymorphism, the C++ compiler may ensure that we do not get typing problems eventually.

[5] We are aware of a solution to encode static class hierarchies that is different to the one presented later on in this paper. However, one drawback of this alternate solution is to duplicate every class: having a class `Bar` in OOP gives rise to a couple of classes in the static hierarchy. To our opinion, this is both counter-intuitive and tedious.

Overloading. In the context of scientific computing, having overloading is crucial. For instance, we expect from the operator "+" to be an over-overloaded function in an algebraic library. Moreover, overloading helps to handle a situation that often arises in scientific libraries: some algorithms have a general implementation but also have different more efficient implementation for particular families of objects. We want to ensure in SCOOP that overloading is as simply manageable as in OOP.

Multimethods. Algorithms are often functions with several input or arguments. Since the source code of an algorithm can also vary with the nature and number of its input, we need multimethods.

Parameter Bounds. Routines of scientific libraries have to be mapped into strongly typed functions. First, this requirement results in a comfort for the users since it prevents them from writing error-prone programs. Second, this requirement is helpful to disambiguate both overloading and multimethod dispatch.

3 Description of SCOOP

3.1 Static Hierarchies

Static hierarchies are meta-hierarchies that result in real hierarchies after various static computations like parameter valuations. With them, we are able to know all types statically hence avoiding the overhead of virtual method resolution. Basically, the core of our static hierarchy system is a generalization of the Barton & Nackman trick [5]. Veldhuizen [54] had already discussed some extensions of this technique and assumed the possibility to apply it to hierarchies with several levels. We effectively managed to generalize these techniques to entire, multiple-level hierarchies.

Our hierarchy system is illustrated in Figure 4. This figure gives an example of a meta-hierarchy, as designed by the developer, and describes the different final hierarchies obtained, according to the instantiated class. The corresponding C++ code is given in Figure 5. This kind of hierarchy gives us the possibility to define abstract classes (class `A`), concrete extensible classes (class `B`), and final classes (class `C`). Non final classes[6] are parameterized by `EXACT` that basically represents the type of the object effectively instantiated. Additionally, any class hierarchy must inherit from a special base class called `Any`. This class factorizes some general mechanisms whose role are detailed later.

Instantiations of abstract classes are prevented by protecting their constructors. The interfaces and the dispatch mechanisms they provide are detailed in Section 3.2.

Extensible concrete classes can be instantiated and extended by subclassing. Since the type of the object effectively instantiated must be propagated through the hierarchy, this kind of class has a double behavior. When such a class `B` is extended and is not the instantiated class, it must propagate its `EXACT` type parameter to its base classes. When

[6] Non final classes are abstract classes or concrete classes that can be extended. Non parameterized classes are necessarily final in our paradigm.

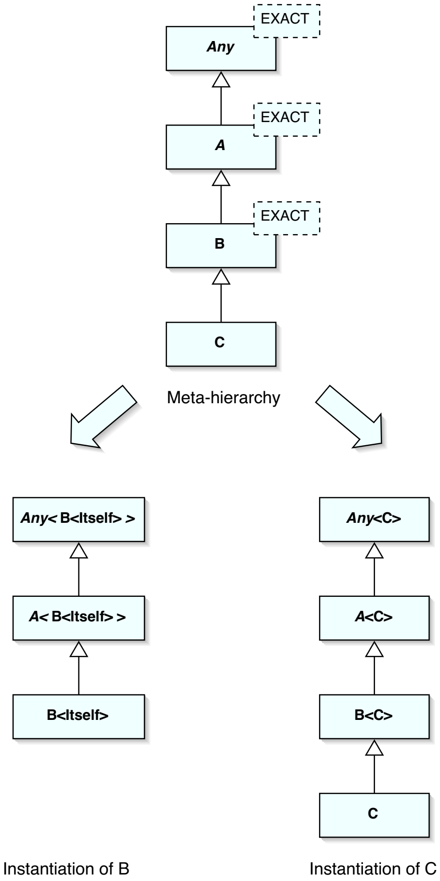

Fig. 4. Static hierarchy unfolding sample
A single meta-hierarchy generates one class hierarchy per instantiable class. Our model can instantiate both leaf classes and intermediate ones. In this example, only `B` and `C` are instantiable, so only the above two hierarchies can be instantiated.
Non final classes are parameterized by `EXACT` which represents the type of the object effectively instantiated. The type `Itself` is used as a terminator when instantiating extensible concrete classes.

```
// Hierarchy apparel

struct Itself
{ };

// find_exact utility macro
#define find_exact (Type) // ...

template <class EXACT>
class Any
{
  // ...
};
```

```
// Hierarchy

// purely abstract class
template <class EXACT>
class A: public Any<EXACT>
{
  // ...
};

// extensible concrete class
template <class EXACT = Itself>
class B: public A<find_exact(B)>
{
  // ...
};

// final class
class C: public B<C>
{
  // ...
};
```

Fig. 5. Static hierarchy sample: C++ code
`find_exact(Type)` mechanism is detailed in Appendix A.1.

it is effectively instantiated, further subclassing is prevented by using the `Itself` terminator as `EXACT` parameter. Then, `B` cannot propagate its `EXACT` parameter directly and should propagate its own type, `B<Itself>`. To determine the effective `EXACT` parameter to propagate, we use a meta-program called `find_exact(Type)` whose principle and C++ implementation are detailed in Appendix A.1. One should also notice that `Itself` is the default value for the `EXACT` parameter of extensible concrete classes. Thus, `B` sample class can be instantiated using the `B<>` syntax.

Itself classes cannot be extended by subclassing. Consequently, they do not need any `EXACT` parameterization since they are inevitably the instantiated type when they are part of the effective hierarchy. Then, they only have to propagate their own types to their parents.

Within our system, any static hierarchy involving n concrete classes can be unfolded into n distinct hierarchies, with n distinct base classes. Effectively, concrete classes instantiated from the same meta-hierarchy will have different base classes, so that some dynamic mechanisms are made impossible (see Section 2.3).

3.2 Abstract Classes and Interfaces

In OOP, abstraction comes from the ability to express class interfaces without implementation. Our model keeps the idea that C++ interfaces are represented by abstract classes. Abstract classes declare all the services their subclasses should provide. The compliance to a particular interface is then naturally ensured by the inheritance from the corresponding abstract class.

Instead of declaring pure virtual member functions, abstract classes define abstract member functions as dispatches to their actual implementation. This manual dispatch is made possible by the `exact()` accessor provided by the `Any` class. Basically, `exact()` downcasts the object to its `EXACT` type made available by the static hierarchy system presented in Section 3.1. In practice, `exact()` can be implemented with a simple **`static_cast`** construct, but this basic mechanism forbids virtual inheritance[7]. Within our paradigm, an indirect consequence is that multiple inheritance implies inevitably virtual inheritance since `Any` is a utility base class common to all classes. Advanced techniques, making virtual and thus multiple inheritance possible, are detailed in Appendix A.2.

An example of an abstract class with a dispatched method is given in Figure 6. The corresponding C++ code can be deduced naturally from this UML diagram. In the abstract class `A`, the method `m(...)` calls its implementation `m_impl(...)`. Method's interface and implementation are explicitly distinguished by using different names. This prevents recursive calls of the interface if the implementation is not defined. Of course, overriding the implementation is permitted. Thanks to the `exact()` downcast, `m_impl(...)` is called on the type of the object effectively instantiated, which is necessarily a subclass of `A`. Thus, overriding rules are respected. Since the `EXACT` type is known statically, this kind of dispatch is entirely performed at compile-time and does not require the use of virtual symbol tables. Method dispatches can be inlined so that they finally come with no run-time overhead.

[7] Virtual inheritance occurs in diamond-shape hierarchies.

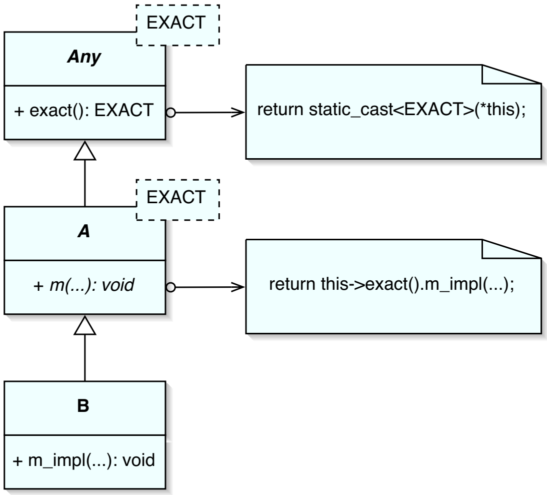
EXACT
Any
+ exact(): EXACT
return static_cast<EXACT>(*this);
EXACT
A
+ m(...): void
return this->exact().m_impl(...);
B
+ m_impl(...): void

Fig. 6. Abstract class and dispatched abstract method sample

3.3 Constraints on Parameters

Using SCOOP, it becomes possible to express constraints on types. Since we have inheritance between classes, we can specify that we only want a subclass of a particular type, thereby constraining the input type. Thus, OOP's ability to handle two different sets of types has been kept in SCOOP, as demonstrated in Figure 7.

Actually, two kinds of constraints are made possible: accept a type and all its subclasses or accept only this type. Both kinds of constraints are illustrated in Figure 8. We have the choice between letting the `EXACT` parameter free to accept all its subclasses, or freezing it (generally to `Itself`) to accept only this exact type.

3.4 Associations

In SCOOP, the implementation of object composition or aggregation is very close to its equivalent in C++ OOP. Figure 9 illustrates the way an aggregation relation is implemented in our paradigm, in comparison with classical OOP. We want a class `B` to aggregate an object of type `C`, which is an abstract class. The natural way to implement this in classical OOP is to maintain a pointer on an object of type `C` as a member of class `B`. In SCOOP, the corresponding meta-class `B` is parameterized by `EXACT`, as explained in Section 3.1. Since all types have to be known statically, `B` must know the effective type of the object it aggregates. A second parameter, `EXACT_C`, is necessary to carry this type. Then, `B` only has to keep a pointer on an object of type `C<EXACT_C>`. As explained in Section 3.3, this syntax ensures that the aggregated object type is a subclass of `C`. This provides stronger typing than the generic programming idioms for aggregation proposed in [20].

```
void foo(A1& arg)
{
    // ...
}

void foo(A2& arg)
{
    // ...
}
```

```
template <class EXACT>
void foo(A1<EXACT>& arg)
{
    // ...
}

template <class EXACT>
void foo(A2<EXACT>& arg)
{
    // ...
}
```

Fig. 7. Constraints on arguments and overloading
Left (classical OOP) and right (SCOOP) codes have the same behavior. Classical overloading rules are applied in both cases. Subclasses of `A1` and `A2` are accepted in SCOOP too; the limitation of GP has been overcome.

```
template <class EXACT>
void foo(A<EXACT>& a)
{
   // ...
}
```

```
void foo(A<Itself>& a)
{
   // ...
}
```

Fig. 8. Kinds of constraints
On the left, `A` and all its subclasses are accepted. On the right, only exact `A` arguments are accepted. As mentioned in section 2.1, contrary to other languages like Ada, C++ cannot make this distinction; this is therefore another restriction overcome by SCOOP.

As for hierarchy unfolding (Section 3.1), this aggregation pattern generates as many versions of B as there are distinct parameters EXACT_C. Each effective version of B is dedicated to a particular subclass of C. Thus, it is impossible to change dynamically the aggregated object for an object of another concrete type. This limitation is directly related to the rejection of dynamic operations, as mentioned in Section 2.3.

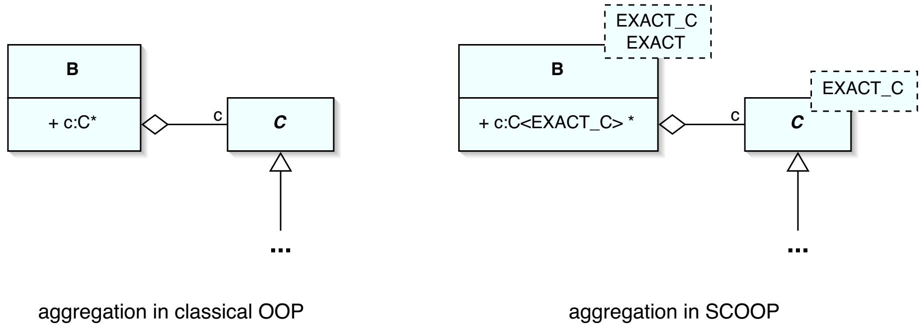

Fig. 9. Comparison of aggregation in OOP and SCOOP

3.5 Covariant Arguments

Covariant parameters may be simulated in C++ in several ways. It can be done by using a **dynamic_cast** to check and convert at run-time the type of the argument. This method leads to unsafe and slower programs. Statically checked covariance has already been studied using templates in Surazhsky and Gil [49]. Their approach was rather complex though, since their typing system was weaker.

Using SCOOP, it is almost straightforward to get statically checked covariant parameters. We consider an example with images and points in 2 and 3 dimensions to illustrates argument covariance. Figure 10 depicts a UML diagram of our example. Since an Image2d can be seen as an Image, it is possible to give a Point3d (seen as a Point) to an Image2d. This is why classical OO languages either forbid it or perform dynamic type checking when argument covariance is involved.

Figure 11 details how this design would be implemented in SCOOP. This code works in three steps:

– Take a Point<P> argument in Image::set and downcast it into its exact type P. Taking a Point<P> argument ensures that P is a subclass of Point at this particular level of the hierarchy.
– Lookup set_impl in the exact image type. Since the point argument has been downcasted towards P, methods accepting P (and not just Point<P>) are candidate.
– In SCOOP, since method dispatch is performed at compile-time, argument covariance will be checked statically. The compilation fails if no method accepting the given exact point type is available.

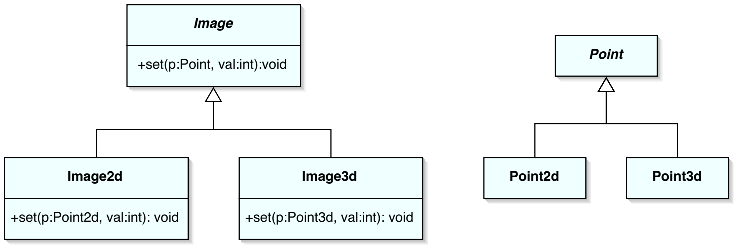
Image
+set(p:Point, val:int):void
Image2d
+set(p:Point2d, val:int): void
Image3d
+set(p:Point3d, val:int): void
Point
Point2d
Point3d

Fig. 10. Argument covariance example in UML

Finally, we have effectively expressed argument covariance. Points have to conform to `Point` at the level of `Image`, and to `Point2d` at the level of `Image2d`.

3.6 Polymorphic **`typedef`**s

In this section we show how we can write virtual **`typedef`**s (we also call them polymorphic **`typedef`**s) in C++. From a base class we want to access **`typedef`**s defined in its subclasses. Within our paradigm, although base classes hold the type of their most derived subclass, it is not possible to access fields of an incomplete type. When a base class is instantiated, its `EXACT` parameter is not completely constructed yet because base classes have to be instantiated before subclasses. A good solution to cope with this issue is to use traits [36, 54]. Traits can be defined on incomplete types, thereby avoiding the infinite recursion.

The overall mechanism is described in Figure 12. To allow the base class to access **`typedef`**s in the exact class, traits have been defined for the exact type (`image_traits`). To ensure correct typedef inheritance, we create a hierarchy of traits which reproduces the class hierarchy. Thus, **`typedef`**s are inherited as if they were actually defined in the class hierarchy. As for argument covariance, virtual **`typedef`**s are checked statically since method dispatch is performed at compile-time. The compilation fails if a wrong point type is given to an `Image2d`.

There is an important difference between classical virtual types and our virtual **`typedef`**s. First, the virtual **`typedef`**s we have described are not constrained. The `point_type` virtual **`typedef`** does not have to be a subclass of `Point`. It can be any type. It is possible to express a subclassing constraint though, by checking it explicitly using a meta-programming technique detailed in Appendix A.3.

One should note that in our paradigm, when using **`typedef`**s, the resulting type is a single type, not a class of types (with the meaning of Section 3.3). A procedure taking this type as argument does not accept its subclasses. For instance, a subclass `SpecialPoint2d` of `Point2d` is not accepted by the `set` method. This problem is due to the impossibility in C++ to make **`template typedef`**s, thus we have to bound the exact type of the class when making a **`typedef`** on it. It is actually possible to overcome this problem by encapsulating open types in boxes. This is not detailed in this paper though.

```
template <class EXACT>
struct Point : public Any<EXACT> {};

template <class EXACT = Itself>
struct Point2d : public Point<find_exact(Point2d)>
{
  // ...
};

template <class EXACT = Itself>
struct Point3d : public Point<find_exact(Point3d)>
{
  // ...
};

template <class EXACT>
struct Image : Any<EXACT>
{
  template <class P>
  void set(const Point<P>& p, int val) {
    // static dispatch
    // p is downcasted to its exact type
    return this->exact().set_impl(p.exact(), val);
  }
};

template <class EXACT = Itself>
struct Image2d : public Image<find_exact(Image2d)>
{
  template <class P>
  void set_impl(const Point2d<P>& p, int val) {
    // ...
  }
};

int main() {
  Image2d<> ima;
  ima.set(Point2d<>(), 42); // ok
  ima.set(Point3d<>(), 51); // fails at compile-time
}
```

Fig. 11. Argument covariance using SCOOP
Compilation fails if the compiler cannot find an implementation of `set_impl` for the exact type of the given point in `Image2d`.

```
// Point, Point2d and Point3d

// A forward declaration is enough to define image_traits
template <class EXACT> struct Image;

template <class EXACT> struct image_traits;

template <class EXACT>
struct image_traits < Image<EXACT> >
{
  // default typedefs for Image
};

template <class EXACT>
struct Image : Any<EXACT>
{
  typedef typename image_traits<EXACT>::point_type point_type;

  void set(const point_type& p, int val) {
    this->exact().set_impl(p, val);
  }
};

// Forward declaration
template <class EXACT> struct Image2d;

// image_traits for Image2d inherits from image_traits for Image
template <class EXACT>
struct image_traits < Image2d<EXACT> >
  : public image_traits <Image <find_exact(Image2d)> >
{
  // We have to specify a concrete type, we cannot write:
  // typedef template Point2d point_type;

  typedef Point2d< Itself > point_type;
  // ... other default typedefs for Image2d
};

template <class EXACT = Itself>
struct Image2d : public Image<find_exact(Image2d)>
{
  // ...
};

int main() {
  Image2d<> ima;
  ima.set(Point2d<>(), 42); // ok
  ima.set(Point3d<>(), 51); // fails at compile-time
}
```

Fig. 12. Mechanisms of virtual `typedef`s with SCOOP

3.7 Multimethods

Several approaches have been studied to provide multimethods in C++, for instance Smith [45], which relies on preprocessing.

In SCOOP, a multimethod is written as a set of functions sharing the same name. The dispatching is then naturally performed by the overloading resolution, as depicted by Figure 13.

```
template <class I1, class I2>
void algo2(Image<I1>& i1, Image<I2>& i2);

template <class I1, class I2>
void algo2(Image2d<I1>& i1, Image3d<I2>& i2);

template <class I1, class I2>
void algo2(Image2d<I1>& i1, Image2d<I2>& i2);

// ... other versions of algo2

template <class I1, class I2>
void algo1(Image<I1>& i1, Image<I2>& i2)
{
  // dispatch will be performed on the exact image types
  algo2(i1.exact(), i2.exact());
}
```

Fig. 13. Static dispatch for multi-methods
`algo1` downcasts `i1` and `i2` into their exact types when calling `algo2`. Thus, usual overloading rules will simulate multimethod dispatch.

4 Conclusion and Perspectives

In this paper, we described a proposal for a Static C++ Object-Oriented Programming (SCOOP) paradigm. This model combines the expressiveness of traditional OOP and the performance gain of static binding resolution thanks to generic programming mechanisms. SCOOP allows developers to design OO-like hierarchies and to handle abstractions without run-time overhead. SCOOP also features constrained parametric polymorphism, argument covariance, polymorphic **`typedef`**s, and multimethods for free.

Yet, we have not proved that resulting programs are type safe. The type properties of SCOOP have to be studied from a more theoretical point of view. Since SCOOP is static-oriented, object types appear with great precision. We expect from the C++ compiler to diagnose most programming errors. Actually, we have the intuition that this kind of programming is closely related to the *matching* type system of Bruce et al. [11]. In addition, functions in SCOOP seem to be f-bounded [12].

The main limitations of our paradigm are common drawbacks of the intensive use of templates:

- closed world assumption;
- heavy compilation time;
- code bloat (but we trade disk space for run-time efficiency);
- cryptic error messages;
- unusual code, unreadable by the casual reader.

The first limitation prevents the usage of separated compilation and dynamic libraries. The second one is unavoidable since SCOOP run-time efficiency relies on the C++ capability of letting the compiler perform some computations. The remaining drawbacks are related to the convenience of the *core developer*, i.e. the programmer who designs the hierarchies and should take care about core mechanisms. Cryptyc error messages can be helped by the use of structural checks mentionned in Section 2.4, which are not incompatible with SCOOP.

This paradigm has been implemented and successfully deployed in a large scale project: Olena, a free software library dedicated to image processing [37]. This library mixes different complex hierarchies (images, points, neighborhoods) with highly generic algorithms.

Although repelling at first glance, SCOOP can be assimilated relatively quickly since its principles remain very close to OOP. We believe that SCOOP and its collection of constructs are suitable for most scientific numerical computing projects.

A Technical Details

A.1 Implementation of find_exact

The find_exact mechanism, introduced in Section 3.1, is used to enable classes that are both concrete and extensible within our static hierarchy system. This kind of class is parameterized by EXACT: the type of the object effectively instantiated. Contrary to abstract classes, concrete extensible classes cannot propagate directly their EXACT parameter to their parents, as explained in Section 3.1. A simple utility macro called find_exact is necessary to determine the EXACT type to propagate. This macro relies on a meta-program, FindExact, whose principle is described in Figure 14. and the corresponding C++ code is given in Figure 15.

```
FindExact($Type$, $EXACT$)
{
  if $EXACT \neq$ ''$Itself$''
    return $EXACT$;
  else
    return $Type < Itself >$;
}
```

Fig. 14. FindExact mechanism: algorithmic description

```
// default version
template <class T, class EXACT>
struct FindExact
{
  typedef EXACT ret;
};

// version specialized for EXACT=Itself
template <class T>
struct FindExact<T, Itself >
{
  typedef T ret ;
};

// find_exact utility macro
#define find_exact (Type) typename FindExact<Type<Exact>, Exact>::ret
```

Fig. 15. FindExact mechanism: C++ implementation

A.2 Static Dispatch with Virtual Inheritance

Using a **static_cast** to downcast a type does not work when virtual inheritance is involved. Let us consider an instance of EXACT. It is possible to create an Any<EXACT> pointer on this instance. In the following, the address pointed to by the Any<EXACT> pointer is called "the Any address" and the address of the EXACT instance is called the "exact address".

The problem with virtual inheritance is that the Any address is not necessarily the same as the exact address. Thus, even **reinterpret_cast** or **void*** casts will not help. We actually found three solutions to cope with this issue. Each solution has its own drawbacks and benefits, but only one is detailed in this paper.

The main idea is that the offset between the Any address and the exact address will remain the same for all the instances of a particular class (we assume that C++ compilers will not generate several memory model for one given class). The simplest way to calculate this offset is to compute the difference between an object address and the address of an Any<EXACT> reference to it. This has to be done only once per exact class. The principle is exposed in Figure 16.

This method has two drawbacks. First, it requires a generic way to instantiate the EXACT classes, for instance a default constructor. Second, one object per class (not per instance!) is kept in memory. If an object cannot be empty (for example storing directly an array), this can be problematic. However, this method allows the compiler to perform good optimizations. In addition, only a modification of Any is necessary, a property which is not verified with other solutions we found.

A.3 Checking Subclassing Relation

Checking if a subclass of another is possible in C++ using templates. The is_base_and_derived<T,U> tool from the Boost [7] type_traits library performs such a check. Thus, it becomes possible to prevent a class from being instantiated if the virtual types does not satisfy the required subclassing constraints.

B Conditional Inheritance

Static hierarchies presented in Section 3.1 come with simple mechanisms. These parameterized hierarchies can be considered as meta-hierarchies simply waiting for the exact object type to generate real hierarchies. It is generally sufficient for the performance level they were designed for. In order to gain modeling flexibility and genericity, one can imagine some refinements in the way of designing such hierarchies. The idea of the conditional inheritance technique is to adapt automatically the hierarchy according to statically known factors. This is made possible by the C++ two-layer evaluation model (evaluation at compile-time and evaluation at run-time) [30]. In practice, this implies that the meta-hierarchy comes with static mechanisms to discriminate on these factors and to determine the inheritance relations. Thus, the meta-hierarchy can generate different final hierarchies through these variable inheritance links.

To illustrate the conditional inheritance mechanism, we introduced a UML-like symbol that we called an *inheritance switch*. Figure 18 gives a simple use case. This

```
template <class EXACT>
struct Any
{
  //  exact_offset  has been computed  statically
  // A good compiler can optimize  this code and avoid any run-time overhead
  EXACT& exact() {
    return *(EXACT*)((char*)this - exact_offset );
  }

private:
  static const int  exact_offset ;
  static const EXACT exact_obj;
  static const Any<EXACT>& ref_exact_obj;
};

//  Initialize  an empty object
// Require a default  constructor  in EXACT
template <class EXACT>
const EXACT Any<EXACT>::exact_obj = EXACT();

// Upcast EXACT into Any<EXACT>
template <class EXACT>
const Any<EXACT>& Any<EXACT>::ref_exact_obj = Any<EXACT>::exact_obj;

// Compute the  offset
template <class EXACT>
const int Any<EXACT>::exact_offset =
  (char*)(&Any<EXACT>::ref_exact_obj)
   - (char*)(&Any<EXACT>::exact_obj);
```

Fig. 16. One method to handle virtual inheritance
The offset between the `Any` address and the address of the `EXACT` class is computed once by using a static object. Since everything is static and const, the compiler can optimize and remove the cost of the subtraction.

```
// ...

template <bool b>
struct type_assert
{};

template <>
struct type_assert <true>
{
  typedef void ret ;
};

#define ensure_inheritance (Type, Base)          \
typedef typename                                 \
   type_assert <                                 \
      is_base_and_derived <Base, Type>::value   \
   >:: ret ensure_##Type

template <class EXACT>
struct Image : Any<EXACT>
{
  typedef typename image_traits<EXACT>::point_type point_type;

   // Will not compile if point_type is not a Point since ret
   // is not defined if the assertion fails .
   ensure_inheritance ( point_type , Point<point_type>);
};

// ...
```

Fig. 17. Specifying constraints on virtual types

example introduces an image hierarchy with a concrete class whose inheritance is conditional: `SpecialImage`. `SpecialImage` is parameterized by an unsigned value `Dim`. We want this class to inherit from `Image2d` or `Image3d` depending on the value of `Dim`. `SpecialImage`'s inheritance is thus represented by an inheritance switch. Figure 19 presents the corresponding C++ code. The inheritance switch is implemented by the `ISwitch` trait parameterized by the dimension value. Its specialization on 2 (resp. 3) defines `Image2d` (resp. `Image3d`) as result type. Finally, `SpecialImage<Dim>` only has to inherit from `ISwitch<Dim>`'s result type.

The factors on which inheritance choices are made are necessarily static values. This includes types, provided by $typedef$s or parameterization, as constant integer values. The effective factors are not necessarily directly available data but can be deduced from static pieces of information. Trait structures can then be used to perform more or less complex information deductions. One should also note that the discriminative factors must be accessible while defining the hierarchy. This implies that these factors must be independent from the hierarchy or externally defined. In practice, class-related factors can be made available outside the hierarchy thanks to trait structures and polymorphic $typedef$s (see Section 3.6).

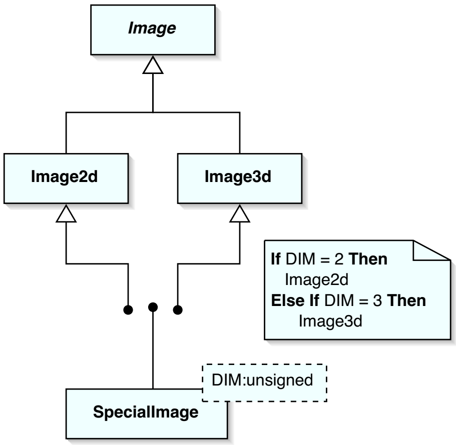

Fig. 18. Simple conditional inheritance sample: UML-like description.

Conditional inheritance mechanisms become particularly interesting when objects are defined by several orthogonal properties. A natural way to handle such a modeling problem is to design a simple sub-hierarchy per property. Unfortunately, when defining the final object, the combinatorial explosion of cases usually implies a multiplication of the number of concrete classes. Figure 20 illustrates an extension of the previous image hierarchy, with more advanced conditional inheritance mechanisms. We extended the image hierarchy with two classes gathering data-related properties, `ColorImage` and `GrayScaleImage`. The hierarchy is now split into two parallel sub-hierarchies.

```
class Image
{
  // ...
};

class Image2d: public Image
{
  // ...
};

class Image3d: public Image
{
  // ...
};
```

```
template <unsigned Dim>
struct ISwitch;

template <>
struct ISwitch<2>
{
  typedef Image2d ret;
};

template <>
struct ISwitch<3>
{
  typedef Image3d ret;
};

template <unsigned Dim>
class SpecialImage
  : public ISwitch<Dim>::ret
{
  // ...
};
```

Fig. 19. Simple conditional inheritance sample: C++ code

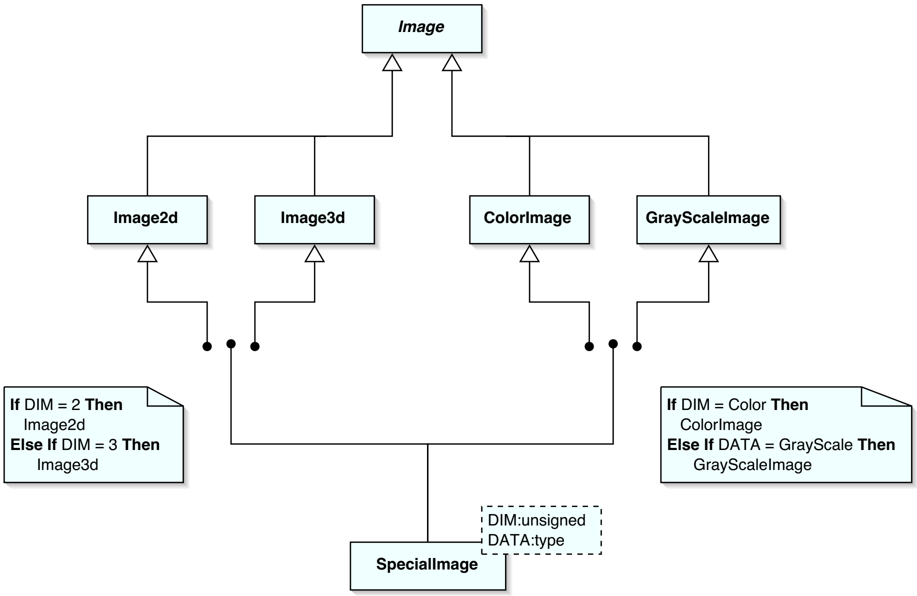

Fig. 20. Conditional inheritance: multiple orthogonal factors.

The first one focuses on the dimension property while the second one focuses on the image data type. The problem is then to define images that gather dimension- and data-related properties without multiplying concrete classes. The idea is just to implement a class template `SpecialImage` parameterized by the dimension value and the data type. Combining conditional and multiple inheritance, `SpecialImage` inherits automatically from the relevant classes. This example introduces the idea of a programming style based on object properties. A `SpecialImage` instance is only defined by its properties and the relevant inheritance relations are deduced automatically.

Finally, mixing conditional inheritance mechanism with other classical and static programming techniques results in powerful adaptive solutions. This work in progress has not been published yet.

Bibliography

[1] International standard: Programming language – C++. ISO/IEC 14882:1998(E), 1998.

[2] G. Aigner and U. Hölzle. Eliminating virtual function calls in C++ programs. In *In the Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP)*, volume 1098 of *Lecture Notes in Computer Science*, pages 142–167. Springer-Verlag, 1996.

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *In the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 324–341, 1996.

[5] J. Barton and L. Nackman. *Scientific and engineering C++*. Addison-Wesley, 1994.

[6] G. Baumgartner and V. F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, January 1997.

[7] Boost. Boost libraries, 2003. URL `http://www.boost.org`.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *In the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[9] K. B. Bruce, A. Fiech, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems (ToPLAS)*, 25(2):225–290, March 2003.

[10] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *In the Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.

[11] K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good ”match” for object-oriented languages. In *In the Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127, Jyväskylä, Finland, 1997. Springer-Verlag.

[12] P. S. Canning, W. R. Cook, W. L. Hill, J. C. Mitchell, and W. G. Olthoff. F-bounded polymorphism for object-oriented programming. In *In the Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 73–280, London, UK, September 1989. ACM.

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

[14] C. Chambers, J. Dean, and D. Grove. Wholeprogram optimization of object-oriented languages. Technical Report UW-CSE-96-06-02, University of Washington, Department of Computer Science and Engineering, June 1996.

[15] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 125–135, San Francisco, California, USA, January 1990. on l'a pas.

[16] J. Coplien. *Curiously Recurring Template Pattern*. In [**?**].

[17] K. Czarnecki and U. Eisenecker. *Generative programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *In the Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, SIGPLAN Notices 31(10), pages 306–323, 1996.

[19] A. Duret-Lutz. Expression templates in Ada. In *In the Proceedings of the 6th International Conference on Reliable Software Technologies, Leuven, Belgium, May 2001 (Ada-Europe)*, volume 2043 of *Lecture Notes in Computer Science*, pages 191–202. Springer-Verlag, 2001.

[20] A. Duret-Lutz, T. Géraud, and A. Demaille. Design patterns for generic programming in C++. In *In the Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 189–202, San Antonio, Texas, USA, January-February 2001. USENIX Association.

[21] G. Furnish. Disambiguated glommable expression templates. *Computers in Physics*, 11(3):263–269, 1997.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns – Elements of reusable object-oriented software*. Professional Computing Series. Addison Wesley, 1995.

[23] T. Géraud, A. Duret-Lutz, and A. Adjaoute. Design patterns for generic programming. In M. Devos and A. Rüping, editors, *In the Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP'2000)*. UVK, Univ. Verlag, Konstanz, July 2000.

[24] T. Géraud, Y. Fabre, and A. Duret-Lutz. Applying generic programming to image processing. In M. Hamsa, editor, *In the Proceedings of the IASTED International Conference on Applied Informatics – Symposium Advances in Computer Applications*, pages 577–581, Innsbruck, Austria, February 2001. ACTA Press.

[25] J. Järvi and G. Powell. The lambda library: Lambda abstraction in C++. In *In the Proceedings of the 2nd Workshop on Template Programming (in conjunction with OOPSLA)*, Tampa Bay, Florida, USA, October 2001.

[26] M. Jazayeri, R. Loos, and D. Musser, editors. *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, 2000. Springer-Verlag.

[27] A. Langer. Implementing design patterns using C++ templates. Tutorial at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2000.

[28] B. Liskov, D. Curtis, M. Day, S. Ghemawhat, R. Gruber, P. Johnson, and A. C. Myers. Theta reference manual. Technical Report 88, Programming Methodology Group, MIT Laboratory for Computer Science, Cambridge, MA, USA, February 1995.

[29] B. Liskov, A. Snyder, R. Atkinson, and J. C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[30] F. Maes. Program templates: Expression templates applied to program evaluation. In J. Striegnitz and K. Davis, editors, *In the Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL; in conjunction with PLI)*, number FZJ-ZAM-IB-2003-10 in John von Neumann Institute for Computing (NIC), Uppsala, Sweden, August 2003.

[31] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[32] B. Meyer. Genericity versus inheritance. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Aplications (OOPSLA)*, pages 391–405, Portland, OR, USA, 1986.

[33] B. Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[34] S. Meyers. How non-member functions improve encapsulation. 18(2):44–??, Feb. 2000. ISSN 1075-2838.

[35] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for java. In *In the Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 132–145, Paris, France, January 1997.

[36] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5): 32–35, June 1995.

[37] Olena. Olena image processing library, 2003. URL `http://olena.lrde.epita.fr`.

[38] oonumerics. Scientific computing in object-oriented languages, 2003. URL `http://www.oonumerics.org`.

[39] Y. Régis-Gianas and R. Poss. On orthogonal specialization in C++: Dealing with efficiency and algebraic abstraction in Vaucanson. In J. Striegnitz and K. Davis, editors, *In the Proceedings of the Parallel/High-performance Object-Oriented Scientific Computing (POOSC; in conjunction with ECOOP)*, number FZJ-ZAM-IB-2003-09 in John von Neumann Institute for Computing (NIC), Darmstadt, Germany, July 2003.

[40] X. Rémy and J. Vouillon. On the (un)reality of virtual types. URL `http://pauillac.inria.fr/~remy/work/virtual/`. March 2000.

[41] U. P. Schultz. Partial evaluation for class-based object-oriented languages. In *Program as Data Objects: International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 2001, Proceedings*, volume 2053 of *Lecture Notes in Computer Science*, pages 173–198. Springer-Verlag, 2001.

[42] J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[43] Y. Smaragdakis and D. Batory. Mixin-based programming in C++. In *In the Proceedings of the 2nd International Conference on Generative and Component-based Software Engineering (GCSE)*, pages 464–478. tranSIT Verlag, Germany, October 2000.

[44] Y. Smaragdakis and B. McNamara. FC++: Functional tools for object-oriented tasks. *Software - Practice and Experience*, 32(10):1015–1033, August 2002.

[45] J. Smith. C++ & multi-methods. *ACCU spring 2003 conference*, 2003.

[46] A. Stepanov, M. Lee, and D. Musser. *The C++ Standard Template Library*. Prentice-Hall, 2000.

[47] J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *In the Proceedings of the 1st Workshop on Template Programming*, Erfurt, Germany, October 2000.

[48] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[49] V. Surazhsky and J. Y. Gil. Type-safe covariance in C++, 2002. URL `http://www.cs.technion.ac.il/~yogi/Courses/CS-Scientific-Writing/examples/paper/main.pdf`. Unpublished.

[50] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, Jyväskylä, Finland, June 1997. Springer-Verlag.

[51] K. K. Thorup and M. Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In R. Guerraoui, editor, *In the Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 186–204, Lisbon, Portugal, June 1999. Springer-Verlag.

[52] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.

[53] T. Veldhuizen. *Expression Templates*, pages 475–487. In [**?**].

[54] T. L. Veldhuizen. Techniques for scientific C++, August 1999. URL `http://extreme.indiana.edu/~tveldhui/papers/techniques/`.

[55] T. L. Veldhuizen and A. Lumsdaine. Guaranteed optimization: Proving nullspace properties of compilers. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*, pages 263–277. Springer-Verlag, 2002.

[56] Xt. A bundle of program transformation tools. Available on the Internet, 2003. URL `http://www.program-transformation.org/xt`.

[57] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *In the Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, volume 32 of *Issue 10*, pages 125–141, Athlanta, GA, USA, October 1997.

Object-Model Independence via Code Implants

Michał Cierniak[1], Neal Glew[2], Spyridon Triantafyllis[3], Marsha Eng[2],
Brian Lewis[2], and James Stichnoth[2]

[1] Microsoft Corporation
[2] Microprocessor Technology Lab, Intel Corporation
[3] Department of Computer Science, Princeton University

Abstract. Managed runtime environments, such as those that execute Java or CLI programs, require close cooperation between the virtual machine (VM) and user code, which is usually compiled by a just-in-time compiler (JIT). Although operations such as field access, virtual method dispatch, and type casting depend on VM data structures, having user code call the VM for these operations is very inefficient. Thus, most current JITs directly generate code for these operations based on assumptions about the VM's implementation. Although this design offers good performance, it locks the VM and the JIT together, which makes modifications difficult. In addition, it is hard to experiment with new algorithms and strategies, which may ultimately hurt performance. Finally, extending a runtime platform to support new programming paradigms may require reimplementing the entire system.

We propose a mechanism that allows VMs to implant code into JIT-compiled code. This mechanism allows a strict interface between the JIT and the VM without sacrificing performance. Among other things, it isolates most programming-paradigm aspects within the VM, thus greatly facilitating multiparadigm support. This paper presents our system for code implants, gives an early evaluation of it, and describes how it could be used to implement several programming-paradigm extensions to Java with relatively little implementation effort.

1 Introduction

The *Open Runtime Platform* (ORP [6, 1]) is a high-performance implementation of a virtual machine for Java[4] [15] and the Common Language Infrastructure (CLI [10]). ORP supports Java and CLI with essentially the same implementation, with only subsets of the implementation being Java or CLI specific. This dual support of both CLI and Java is, as far as we know, unique among virtual machines. It also raises the question of whether we can extend ORP to support other programming paradigms.

ORP uses interfaces to partition its implementation into several well-defined modules: the core virtual machine (VM), the just-in-time compiler (JIT), and the garbage collector (GC). ORP's modular design facilitates experimentation, and allows using multiple JIT and GC implementations with the same core VM. To date we have used seven different JITs (see, *e.g.*, [2, 7, 5, 1]) and five different GCs.

However, portions of ORP's current interfaces sacrifice cleanliness for performance. For instance, the JIT can implement a type-inclusion test either as a call to the VM

[4] Other brands and names are the property of their respective owners.

or as an inlined instruction sequence. While a call-based sequence is independent of the VM's implementation, its performance is slower. An inlined sequence can be very fast (*e.g.*, through the use of Cohen's type displays [8]), but it relies on specific VM data structures and may be incorrect if the VM data structures change. Furthermore, some code sequences are so performance-sensitive that the VM does not provide a function to call, and instead the JIT must rely on specific data structures. Virtual method dispatch is an example. ORP uses vtables for virtual method dispatch, and every JIT must understand this implementation and generate vtable-based code.

Many code sequences assume that ORP only supports single-inheritance languages like Java and CLI, and we never considered abstracting these sequences. For example, an upcast is a no-op in ORP, but common implementations of languages with multiple inheritance (*e.g.*, C++) require that an upcast add an offset to the object pointer. Similarly, for a field access in Java, the JIT can generate the address of the field by adding a compile-time constant to the object pointer. In languages with dependent multiple inheritance, an indirection through an *index table* may be required (see, *e.g.*, Grune *et al.* [13, Section 6.2.9.6]).

For ORP to support other languages such as Python or Scheme, we would have to change ORP's object model to include, for instance, multiple inheritance and possibly multi-methods. This would make its current use of vtables and type displays inappropriate, and would make today's ORP JITs generate incorrect code.

We would also like to experiment with supporting new programming paradigms in ORP. Due to the broad acceptance of the Java language, several Java extensions have been proposed to support additional programming paradigms. These extensions include both standard [14] and dynamic [4, 17] aspect-oriented programming, multiple inheritance, mixin-type inheritance [11], and aspects of functional programming [16, 20]. With the exception of standard aspect-oriented programming, only proof-of-concept implementations exist for the other extensions. More realistic implementations do not exist because of the great difficult of extending most existing Java systems. Even today's ORP would require changes to multiple components including the JITs to support many of these extensions.

This paper proposes a system for the VM to specify, in a CPU-independent fashion, low-level instruction sequences (*stubs*) that the JIT can then inline into the code it emits. This system, which includes the LIL language, simplifies the generation of the intricate code sequences needed to implement various language operations like a virtual method dispatch, a type-inclusion test, or a type cast. We argue that it can also support new object models and programming paradigms. It abstracts away details that depend on the object model or programming paradigm, and make JITs oblivious to the details of their implementation. We further argue that these benefits will be possible without sacrificing the performance of today's ORP. It has the additional benefit of simplifying maintenance for multiple CPU architectures.

This paper presents our results to date. We describe the LIL language in Section 2. To make our proposal concrete, we describe in some detail how to implement one operation, downcasts, using LIL in Section 3. We discuss how the VM and JIT interoperate to inline this code sequence into JIT-generated code, and present the results of an experiment to evaluate the performance impact from inlining this operation. Section 4 gives an overview of using this system for adding new programming paradigms to ORP with only modest implementation effort.

2 LIL

We designed a language called LIL[5] to express low-level code to be inlined in a CPU-independent way. This section gives a brief overview of LIL.

Here is an example of a LIL stub that invokes a virtual method on an object.

```
entry 0:managed:ref,g4,f4:g1;
locals 1;
ld l0,[i0+0:pint];
ld l0,[l0+32:pint];
in2out managed:g1;
call l0;
ret;
```

This stub is compiled into code that acts like a function. The stub's `entry` line states that it is called using the managed-code calling convention (i.e., the VM-specified convention for calling JIT-generated code) with three arguments, and that it returns a result. The arguments are of type `ref` (reference to an object in the heap), `g4` (32-bit general-purpose value), and `f4` (32-bit floating-point value), and the result is of type `g1` (8-bit general-purpose value). (The "0" reflects a low-level implementation detail that is beyond the scope of this paper.) The rest of the stub consists of the instructions that are to be executed when the stub is called.

- The `locals 1;` instruction declares a single local variable.
- The `ld l0,[i0+0:pint]` instruction loads a `pint` (pointer-sized general purpose value, often used for pointers that are not objects in the heap) from the address given by `i0` (the first argument) into `l0` (the first local)—in this example, this pointer points to the vtable for the object whose method is being invoked.
- The second `ld` instruction loads a `pint` from the address given by `l0` plus 32—in this example, this is the entry in the vtable for the method being invoked.
- The third instruction (`in2out managed:g1`) sets up for a call; in particular, it copies the arguments into an output area, and declares that the call will use the managed-code calling convention and return a `g1` into the implicit variable `r`.
- The `call l0` instruction calls the address in `l0` and sets the variable `r` to the value returned.
- The final `ret` instruction returns. The current value of `r` is the value returned by the stub.

Notice that the stub implicitly makes a tail call. LIL has an explicit way to make a tail call that is optimized by the code generator. The above stub could also be expressed as:

[5] LIL stands for Low-level Intermediate Language, and its pronunciation suggests its "little"ness or lightweight nature.

```
entry 0:managed:ref,g4,f4:g1;
locals 1;
ld l0,[i0:pint+0:pint];
ld l0,[l0+32:pint];
tailcall l0;
```

All LIL variables and operations are typed by a simple type system. The type system makes just enough distinctions to know the width of values and where they should be placed in a given calling convention. For example, the type system distinguishes between floating-point and general-purpose values but not between signed and unsigned. In addition, the type system distinguishes various kinds of pointers (e.g., pointers to heap objects versus pointers to fixed VM data structures), because in the future we may want the LIL code generator to be able to enumerate GC roots on LIL activation frames.

A LIL stub executes with an activation frame on the stack. Conceptually, this activation frame is divided into a number of areas that can vary in size and type across the execution of the stub. For our purposes, the areas are inputs, locals, outputs, and return. The inputs initially hold the arguments parsed by the caller, but they can change by assignment. Their number and type is fixed across the stub. The locals hold values local to the stub. Their number is determined by `locals` instructions, and their types are determined by a simple flow analysis. The outputs hold values passed to functions called by the stub. Their number and types are determined by `in2out` and `out` instructions. These instructions set up an output area and assign to the outputs, and then a `call` instruction performs the actual call. The return is a single location that is present following a `call` instruction or whenever an assignment is made to it; its type is determined by a flow analysis. Each input, local, output, and return is a LIL variable, and are referred to using the names `i0`, `i1`, . . . , `l0`, `l1`, . . . , `o0`, `o1`, . . . , and `r`, respectively.

LIL's instructions include arithmetic operations, loads, stores, conditional and unconditional jumps, calls, and returns. They are summarized in Table 1. An operand `o` is either a LIL variable or an immediate value. The address part of load, store, and increment instructions can include a base variable, a scaled index variable, and an immediate offset; the scale can be one, two, four, or eight. This format was chosen to easily take advantage of instructions and addressing modes of the IA32 architecture, the Itanium® Processor Family (IPF) architecture, and other architectures. The address also includes the type of the value being loaded, stored, or incremented. The conditions in a conditional jump are standard comparisons (equal, not equal, less, etc.) between two operands.

LIL also includes some constructs specific to our virtual machine, such as accessing thread-local data for synchronization or object allocation. However, these do not concern this paper and will not be discussed further.

The LIL system has two parts: a parser and a code generator. The parser takes a C string as input and produces an intermediate representation (IR) of LIL instructions. The parser includes a `printf`-like mechanism for injecting runtime constants such as addresses of functions or fixed VM data structures. The code generator takes the LIL IR as input and produces machine instructions for a particular architecture.

Table 1. LIL General-Purpose Instructions

Category	LIL syntax	Description
Declarations	`:label;`	
	`locals n;`	
	`in2out cc:rettype;`	
	`out sig;`	
Arithmetic	`v = o;`	Move
	`v = uop o;`	Unary
	`v = o1 op o2;`	Binary
Memory access	`ld v, addr;`	Load
	`st addr, o;`	Store
	`inc addr;`	Increment
Calls	`call o;`	Normal call
	`tailcall o;`	Tail call
	`call.noret o;`	Call that does not return
	`ret;`	
Branches	`jc cond, label;`	Conditional branch
	`j label;`	Unconditional branch

3 Subtype Tests

To illustrate how the VM can use LIL to insulate JITs from details of the object model and type data structures, this section considers subtype tests. Both Java and CLI include bytecodes like `checkcast` and `instanceof` that test whether an object is in a particular type. The typical implementation dereferences the object to obtain its class's data structure, and then tests whether this class is a subtype of the target type. The naive implementation of this subtype test is to traverse superclass, interface, and array element-type pointers searching for the target type. In contrast, ORP uses Cohen's type display technique [8]. In practice, this implementation is much faster than the naive implementation, and improves overall application performance [3]—even more so when the JIT inlines the fast path into its generated code. However, the code to be inlined is heavily dependent upon the details of Cohen's techniques and the details of the VM's data structures. This example is an ideal case for showing how knowledge can be isolated in the VM without sacrificing performance.

3.1 Cohen's Type Displays

The basic idea of Cohen's type displays is to store a table of ancestors in the type data structures. In ORP, we store a fixed sized (MAX_DEPTH $- 1$) table in the vtable of each class. For a class at depth d, if $d \leq$ MAX_DEPTH, then the table contains the class's ancestors at level two through d and then NULLs; otherwise, the table contains the class's ancestors at level two through MAX_DEPTH. Note that every class's level-one ancestor is `java.lang.Object` (or `System.Object` in CLI), so we do not need to store this class. Each entry points to the class data structure for the corresponding class (not to the vtable of that class). To test if a class represented by vtable v is a subtype of another class represented by class data structure c, ORP does the following. If c's depth is one (meaning c is `java.lang.Object`), the test succeeds. Otherwise if c's depth d is less than or equal to `MAX_DEPTH`, ORP compares entry $d - 1$ of v's ancestor table with c, and this is the result of the subtype test. Otherwise, ORP falls back to the naive implementation, which is also used if the target type is an interface or array type. Since the performance-critical cases are most often class types within the maximum depth, most of the time a short sequence of instructions is executed.

3.2 Implementation

ORP offers runtime helper functions for subtype test operations, and the JIT may generate calls to the appropriate helper. For `checkcast`, the helper function takes two arguments: the object, and the class data structure for the target type. To call this helper function, the JIT emits instructions corresponding to the following simple LIL stub:

```
entry 0:managed:ref,pint:ref;
tailcall checkcast_helper;
```

Since the JIT knows the target type at compile time, the JIT could obtain better performance by inlining a fast path sequence customized to the target type. To achieve this inlining in ORP without LIL, the JIT would need to know that Cohen's type displays are being used, the location of the vtable within objects, and the location of the ancestor table in vtables. If any of these details change, then the JIT must be changed. If the object model is changed or Cohen's algorithm is replaced by another, perhaps to accommodate multiple inheritance, then the JIT must change accordingly. Prior to developing LIL, ORP did have a JIT that did this customized inlining, so we are able to compare its performance against the LIL version.

To address both the performance issue and the software engineering issue, we modified ORP to use LIL to communicate information from the VM to the JIT without making the JIT dependent on this information. For `checkcast`, this works as follows: The JIT requests from the VM the runtime helper function for `checkcast`, and passes at JIT time the class data structure for the target type. The VM can optionally return a LIL stub, consisting of a customized sequence for that type, to inline into the JIT-generated code. If the type is a class type of less than maximal depth, the LIL stub will be the following (where `d` is the depth, `c` is the class data structure for the type, `ato` is the offset of the ancestor table within a vtable, and `throw_class_cast_exception` is a helper function that throws an appropriate exception):

```
entry 0:managed:ref,pint:ref;
jc i0!=0,nonnull;
r=i0;
ret;
:nonnull;
locals 1;
ld l0,[i0+0:pint];
ld l0,[l0+ato+4*(d-1):pint];
```

```
jc l0!=c,failed;
r=i0;
ret;
:failed;
out managed::void;
call.noret throw_class_cast_exception;
```

(Note that the stub is a two-argument function even though it specialized on its second argument, and that `ato+4*(d-1)` is actually a constant computed by the VM and is not a LIL expression.) The JIT inlines and converts the LIL stub into its own internal IR. For typical compiler IRs, this should be straightforward.

In this new implementation, the JIT is not aware that the type inclusion tests are implemented with type displays. Therefore, if the VM were modified to support multiple inheritance, the same JIT would still work for Java programs without any modifications and with no performance penalty. A new implementation could use a technique more suitable for multiple inheritance like the one described by Vitek *et al.* [19].

The scheme achieves our goal: it allows us to make performance optimizations without making the JIT dependent upon any VM information. All the JIT needs to understand is LIL and how to inline it. The next section evaluates the resulting performance, showing both that the LIL version performs similarly to the customized JIT version and that both versions of the JIT achieve speedup over the unoptimized call version.

3.3 Performance evaluation

As an experiment, we modified ORP's high-performance O3 JIT to include an ad-hoc runtime helper inlining system. This includes both a LIL inliner and custom versions of `checkcast` and `instanceof`. The custom version requires the JIT to have specific knowledge of the type display scheme including details of the VM data structures; the LIL version insulates the JIT from all such details. This section compares the performance of both versions of inlining with doing no inlining. We use the SPEC JVM98 [18] benchmark to perform the comparison.[6] The measurements are taken on a four processor 2.0 GHz Intel® Xeon™ machine with HyperThreading disabled, with 4 GB of RAM, and a heap size of 96 MB.

The results appear in Figure 1. The baseline does no inlining of helper functions; that is, JIT-generated code calls helper functions in the VM (however, these VM helper functions do have a fast path using the type displays). The graph shows the performance improvement of the two versions of inlining over this baseline.

For the most part, there are small but significant performance improvements from inlining the helper functions. The gains from both schemes are in the same order of magnitude, ranging from no improvement on Jack and 5.6% on Db for ad-hoc inlining, and less than 1% on Compress and 6.3% on Db for LIL inlining. These results are encouraging and suggest that code implants offer good performance along with their other benefits.

[6] We use the SPEC benchmarks only as benchmarks to compare the performance of the various techniques within our own VM. We are not using them to compare our VM to any other VM and are not publishing SPEC metrics of any kind.

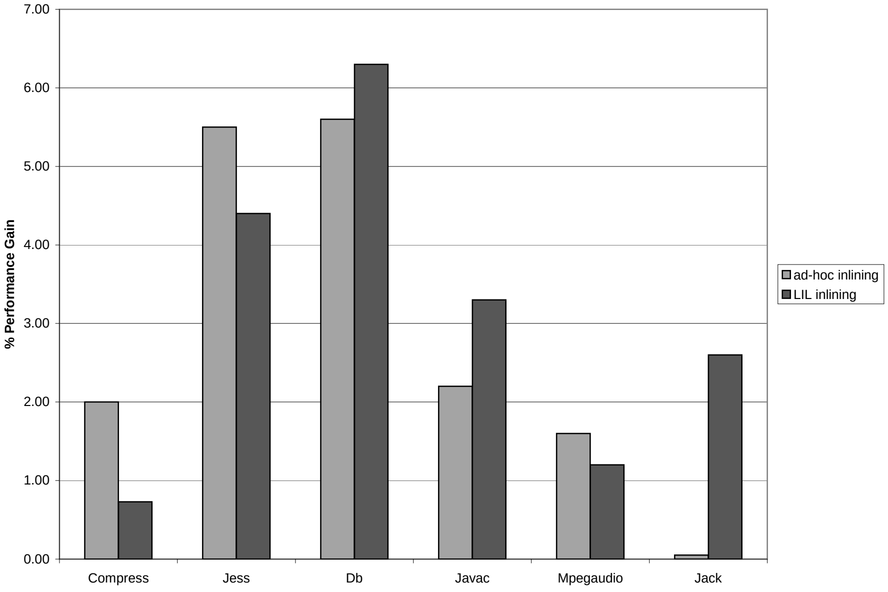

Fig. 1. Performance comparison of inlining on SPEC JVM98.

4 Multiparadigm Support

This section shows how LIL code implants can support extensions to ORP's programming paradigm. As ORP stands today it is not suited for such extensions because some operations are implemented through ad-hoc JIT-generated sequences. In particular, field access, method invocation, downcasts, and upcasts are implemented by the JIT. The first step towards extending ORP's programming paradigms is to require these operations be implemented with LIL code implants. Once this is in place, the modifications required for the extensions discussed in this section are to the VM data structures and to the LIL that the VM provides to the JITs. The JIT implementation is largely unaffected. In our experience, the JIT is often the most complex part of the system and any changes to it are difficult. Therefore, limiting the changes needed to support a new programming paradigm to the core VM greatly simplifies the extension's implementation.

4.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) has been gaining popularity in recent years. Typically AOP extensions to Java are implemented without any changes to the JVM implementation and generate standard class files. AspectJ [9] is a popular example of such a system.

A new class of AOP applications is emerging that require dynamic AOP [4, 17]: the ability to add, modify and remove aspects at runtime. In a typical scenario, the source code is not available and and it cannot be rewritten in an AOP version of Java.

Instead, the system must modify an existing application. The point cuts needed for dynamic AOP usually include method invocations and (sometimes) field accesses. The approaches used in prior work can be grouped into three categories:

- Use the debugger interface to intercept method invocations. The downside of this approach is poor performance and a limited set of point cuts available (*e.g.*, field accesses may not be available).
- Modify the bytecodes via a custom class loader. While this implementation is flexible, its performance is a problem since each potential point cut must be instrumented and a check must be performed at runtime to determine if a given point cut is active. Bytecode instrumentation of every point cut cannot be made as cheap as custom solutions.
- Modify the JIT to emit appropriate code. This approach offers the best performance but it requires potentially cumbersome modifications to the JIT.

The use of LIL allows dynamic AOP to be implemented with the same performance as the latter solution, but without the need to modify the JIT. The VM can use data structures to keep track of dynamic aspects. These data structures can then be consulted to generate the appropriate LIL sequences for each method invocation and field access.

4.2 Multiple Inheritance and Mixins

Changing the type inheritance model affects type instantiation, type casts, field accesses, and method invocation. Since all these actions are implemented through implanted LIL sequences, any changes needed to support a new inheritance model are confined within the VM. The rest of this section describes how multiple inheritance, mixins, and extensible objects can be supported using this system.

Classic Multiple Inheritance The easiest way to implement multiple inheritance is the one followed by most C++ compilers. That is, each base class instance begins at a certain offset from the start of the object. Each base class instance also has its own vtable pointer, which differs from the object's vtable pointer by a certain offset.[7] If this implementation is followed, then only the stubs for type casts need to change. Instead of just returning its argument, an upcast stub would now look as follows:

```
entry 0:managed:ref:ref;
r=i0+BASE_OFF;
ret;
```

where BASE_OFF is the offset of the requested base class's instance within the object. Downcast stubs must change in a similar but opposite way.

[7] This implementation slightly complicates garbage collection, since references now do not necessarily point to the start of an allocated object. This problem can be solved, although the details are too technical to include here.

Mixins As explained by Flatt *et al.* [11], implementing mixin-style inheritance may require “boxing” some references. That is, a mixin-typed reference is implemented as a pointer to a structure that contains a pointer to an actual object as well as field and method translation tables (and possibly other information). These tables map field and method names into the offsets within the object and vtable respectively. Casts from normal class types to a mixin type must create the boxed reference, whereas casts from mixin types to a normal class type must retrieve the underlying object and discard the outer structure. Field accesses and method invocations through mixin references require looking up the field or method name in the translation tables. Clearly the VM can make LIL stubs for casts, field access, and method invocation that match this scheme; we omit the details.

Extensible Objects Some object-oriented languages such as Python allow the user to dynamically add fields to an object at runtime. A simple implementation has a list of dynamic field names and values in each object. If a JIT requests a field not statically declared in a type, then the VM generates a stub to search the dynamic field list of the object. If a requested field is not found in the dynamic field list, the stub can either throw an exception or create the field depending upon the desired language semantics. The VM can also provide the JIT with LIL stubs for dynamic field addition operations. Clearly, optimizations of this simple scheme are also expressible using LIL stubs.

4.3 Functional Programming

It is well known that functions, especially first-class functions in functional programming languages, are equivalent to objects with a single method [12]. Function application becomes invocation of the single method. The VM can present functions and function types to a Java JIT as if they were objects. The method invocation sequence shown in Section 2 results in two loads and a call for a function application. Typical implementations of functional programming languages have only one load and a call. The extra load could degrade performance. To avoid it, the VM could store the code address directly in the function, say immediately after the vtable.[8] Then if a Java JIT requests a virtual dispatch on a function type, the VM can generate a LIL sequence that loads the code pointer directly from the function, such as the following (for a function of no arguments or returns):

```
entry 0:managed:ref:void;
locals 1;
ld l0,[i0+4:pint];
tailcall l0;
```

Functional programming languages also have features like polymorphism, discriminated unions, and lazy evaluation that are quite different from typical object-oriented

[8] In typical implementations of functional programming, there is no vtable, but instead there is a header word used by the garbage collector. In ORP, there is no header word, and instead the information contained in the header word is stored in the vtable. Thus the space requirements are the same.

features. For example, polymorphism in these functional programming languages is related to generics in object-oriented languages. These features might need different object models, field access sequences, and method invocation sequences. As an example of the latter, thunk creation and forcing for lazy evaluation could be hidden in method invocation code that crosses from Java to lazy functional code. We speculate that these differences could be hidden from the JIT using LIL.

5 Conclusion

Extending existing virtual machines to support new object models and programming paradigms is difficult because knowledge about the object model is spread across many components and modules. Such knowledge includes how field accesses, method invocations, down casts, and up casts should be implemented. This paper has shown how to address this problem without sacrificing performance. The solution is to concentrate knowledge of the object model in the core VM component, and to use code implants to inline performance critical operations into JIT-generated code and other components. Then implementing new object models or programming paradigms requires changes to a small part of the system only.

We originally designed LIL to provide CPU-independence and better maintainability of stubs within ORP. We were pleasantly surprised when we realised that it could also be used to implement a code implant system and thus achieve better modularity for ORP with the same performance.

While we have investigated code implants for `checkcast` and `instanceof` in ORP, much more work remains before ORP will be a platform for multiparadigm experiments. Many other opportunities remain for the use of code implants. Future work will explore these possibilities.

Bibliography

[1] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT Compiler: A Dynamic Compiler for Managed Runtime Environments. *Intel Technology Journal*, 7(1), February 2003. Available at `http://intel.com/technology/itj/2003/volume07issue01/art02_starjit/p01_abstract.htm`.

[2] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[3] B. Alpern, A. Cocchi, and D. Grove. Dynamic Type Checking in Jalapeño. *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, April 2001.

[4] J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. *Proceedings of the International Conference on Aspect-Oriented Software Development*, April 2002.

[5] A. Bik, M. Girkar, and M. Haghighat. Experiences with JAVA JIT Optimization. *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, October 1998.

[6] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003. Available at `http://intel.com/technology/itj/2003/volume07issue01/art01_orp/p01_abstract.htm`.

[7] M. Cierniak, G.-Y. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.

[8] N. H. Cohen. Type-extension type test can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4), October 1991.

[9] Eclipse.org. AspectJ Project, 2003. Available at `http://eclipse.org/aspectj`.

[10] ECMA. *Common Language Infrastructure*. ECMA, 2002. Available at `http://www.ecma-international.org/publications/Standards/ecma-335.htm`.

[11] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

[12] N. Glew. Object closure conversion. In A. Gordon and A. Pitts, editors, *3rd International Workshop on Higher-Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, France, Sept. 1999. Elsevier. Available at `http://www.elsevier.nl/locate/entcs/volume26.html`.

[13] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendongen. *Modern Compiler Design*. Wiley, 2000.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[16] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997.

[17] A. Popovici, G. Alonso, and T.Gross. Just In Time Aspects: Efficient Dynamic Weaving for Java. *Proceedings of the International Conference on Aspect-Oriented Software Development*, March 2003.

[18] Standard Performance Evaluation Corporation. SPEC JVM98, 1998. See `http://www.spec.org/jvm98`.

[19] J. Vitek, R. N. Horspool, and A. Krall. Efficient type inclusion tests. *ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*, October 1997.

[20] D. Wakeling. Compiling lazy functional programs for the java virtual machine. *Journal of Functional Programming*, 9(6), November 1999.

The ConS/* Programming Languages

Matthias M. Hölzl

hoelzl@informatik.uni-muenchen.de
Institut für Informatik
Ludwig-Maximilians Universität München
Oettingenstraße 67
80538 München

Abstract. The ConS/* family of languages extends object-functional languages like Common Lisp, Dylan or Scheme with facilities for "non-deterministic" computation and constraint solving. Integration of constraint solving is achieved without changing the evaluation mechanism of the underlying base language; ConS/* achieves close integration between the base language and constraint solving by providing parametric backtracking, first-class constraints and generic built-in operators.

1 Introduction

Languages that offer a single programming paradigm often have a simple conceptual model and nice theoretical properties. On the other hand, experience shows that no single programming paradigm is suitable to solve all programming problems with equal conciseness and clarity.

Currently, many programs are written in object oriented languages since these languages seem to offer a good compromise for most application areas. For certain application domains, however, other paradigms provide significant advantages. For example, Constraint Logic Programming (CLP) is an established declarative programming paradigm that is very well suited to many kinds of optimization problems.

It is therefore not surprising that various languages try to integrate more conventional programming paradigms with constraint-solvers. These languages can roughly be divided into two groups. Languages in the first group modify the argument-passing mechanism of function or method applications: If a function is applied to an argument which is not completely determined, the computation either suspends (the so-called *residuation* principle) or non-deterministically assigns a value to the variable (*narrowing*). Most of these languages are based on a functional core, for example *Curry* [2] or *Babel* [10]. Languages in the second group add constraint-solving extensions as a sub-language to a functional or object oriented language. This approach can be found in *non-deterministic Lisp* [12, 13], as implemented by the *Screamer* package [14], the programming language *Alice* [18, 19], the *Java Constraint Kit (JCK)* [1] or the the *ILOG Solver* and *Optimization Suite* products [6, 5].

The two groups of languages differ markedly in the programming-style that they support naturally. In general, languages in the first group tightly integrate constraint solving into their base language. In general this leads to languages that offer seamless integration between the different programming paradigms. The disadvantage of this approach is that the complex evaluation model of logic programming languages is carried over into the base language and that many features that are essential to object oriented languages (e.g., mutation) are difficult to support. Languages in the second group require the programmer to chose for each subproblem of the program whether to solve it in the base language or in the constraint sublanguage. On the other hand they often offer more possibilities for compiler optimization and, more importantly, a simpler execution model.

We present the *ConS/** family of languages. The languages in the *ConS/** family are based on constraint-functional languages; they retain the simple argument-passing and evaluation model of object-oriented or eager functional languages, but integrate backtracking and constraint-solving as tightly as languages based on residuation or narrowing. This is achieved by adding *"non-deterministic computation"*[1] and *first-class constraints* to the language, and by extending the built-in special-operators and functions to handle constraints in the appropriate manner. Since most facilities in *ConS/** offer a *meta-object protocol* [9], they can be customized and extended by the user. The language facilities described in this paper can be added to different object-oriented languages; for concreteness we focus on *ConS/Scheme* in the rest of the paper.

This paper is organized as follows: In the next section we present an introductory example that shows how the new features can be used to "reverse" the computation of a program. In the third section we give a short introduction to the core language and its object system. We then show very briefly how an object-oriented protocol can be used to specify search and solution strategies, and how the same search-protocol can be used as meta-object protocol to control the backtracking behavior of the language. We proceed to describe how most "built-in" facilities can be expressed using generic-functions and how this approach leads to a natural integration of constraints into the language.

2 An Example Program

To illustrate the constraint mechanism we show how an example program that is written in straightforward object oriented style can be used to compute input arguments that result in a desired outcome of the program. The program is modeled after an example in [11]; a constraint-functional translation that is closer to the original is given in [4]. It computes the balance B of a mortgage with principal P, after T years, if the interest rate is I and the annual repayment is R. Figure 1 shows a Java class that computes the balance of the mortgage, Figure 2 shows the corresponding *ConS/Scheme* program.

This example shows that there are several differences between *ConS/Scheme* and Java. The most obvious difference is that *ConS/Scheme* is written in prefix notation whereas Java uses a more conventional algebraic (infix) syntax. Another difference is that in *ConS/Scheme* methods do not "belong to" classes and have no implicit *this* argument. Instance variables are usually accessed by calling *getter* methods. In principle the

[1] The term "non-deterministic" computation is actually a misnomer since the computation proceeds in a completely determined fashion. Since this usage is common in the CPL-literature we also use it in this paper.

```
class Mortgage {
    private double principal = 10000.0;
    private double interestRate = 0.05;
    private double repayment = 1000.0;

    double balance(int time) {
        if (time >= 0) {
            double result = principal;
            for (int i = 0; i < time; i++) {
                result += result * interestRate - repayment;
            }
            return result;
        }
        else
            throw(new RuntimeException("Negative time?"));
    }
}
```

Fig. 1. Java class for the mortgage example.

```
(define-class <mortgage> (<object>)
  ((principal :init-keyword :principal
              :initform 10000.0
              :accessor principal)
   (interest-rate :init-keyword :interest-rate
                  :initform 0.05
                  :accessor interest-rate)
   (repayment :init-keyword :repayment
              :initform 1000.0
              :accessor repayment)))

(define-method balance ((mortgage <mortgage>) time)
  (if (>= time 0)
        (let ((result (principal mortgage)))
          (dotimes (i time)
            (inc! result (- (* result (interest-rate mortgage))
                            (repayment mortgage))))
          result))
      (error "Negative time?"))
```

Fig. 2. *ConS/Scheme* program for the mortgage example.

techniques developed in this paper could be implemented in a version of Java in which some constraints of the type system are relaxed.

It is clear that the programs are semantically equivalent: Both programs define a class that contains instance variables for the principal, interest rate and repayment of the mortgage and a method *balance* that computes the balance of the mortgage after a certain number of years.

If we call the *ConS/Scheme* function `balance` with known quantities for its arguments, say a mortgage with `principal` $= 100$, `interest-rate` $= 0.05$ and `repayment` $= 10$, and `time` $= 1$ it will compute the balance after one year. The operational behavior of the program is similar to the Java program: `Balance` evaluates the term `(>= time 0)` with `time` bound to 1. This results in the value `true`, therefore evaluation proceeds with the first clause of the `if` statement. This clause binds the variable `result` to the value of the `principal` instance variable of the mortgage. It then binds the loop variable `i` to 0 and evaluates the loop body. After the first evaluation of the body, the condition of the loop is tested again; this time the test fails, therefore evaluation continues after the loop and returns the value of `result` to the caller.

There is one important difference between both programs given here and the original constraint logic version in [11] (apart from the obvious syntactic one): The constraint program describes a *relation* between its arguments while the Java and *ConS/Scheme* programs compute a *function* from their arguments to results.

We can therefore use the constraint-logic version of the program to answer other questions than the balance given a mortgage and a time. If, for example, we want to know the time we have to wait until we have a balance of 95 when starting with a principal of 100, an interest rate of 0.05, and a repayment of 10, the Java program is no longer able to compute an answer while the constraint program answers $T = 1$. In the following paragraphs we show how the *ConS/Scheme* program can answer this question as well: In *ConS/Scheme* the query that asks for a given balance can be written as

```
(begin
 (= (balance mortgage ?T) 95)
 ?T)
```

which is expanded to a program similar to the following:

```
(let* ((t (constraint-variable))
       (x (balance mortgage t)))
  (add-constraint (= x 95))
  (determined-value t))
```

This program computes its result in the following way:

First the variable `t` is bound to a fresh constraint variable T, then the function `balance` is called with an instance of the class `<mortgage>` and the newly created variable T. Since function application is defined in the usual way, we now evaluate the body of the `balance` function with `mortgage` bound to the mortgage instance and `time` bound to the constraint variable T passed as argument. This means that we evaluate the term `(>= time 0)`. At this point we encounter the first difference from the usual functional evaluation model: Comparison functions like `=` or `>=` are defined for arguments which are constraint-variables and their semantics is extended in the following way: If >= is applied to "known" values like numbers or strings, it evaluates to either true or false. If, however, one of its values is a constraint variable it evaluates to a *constraint*, in this case to the constraint $T \geq 0$. The behavior of the special operator if is polymorphic as well: If its argument is a known value it evaluates one of the branches; if its first argument evaluates to a constraint C, the if-operator evaluates *both* branches non-deterministically. In the consequent branch, C is added to the constraint store before the forms in the branch are evaluated, in the alternative branch, $\neg C$ is added to the constraint store.

Proceeding with our example, to evaluate the consequent of the if-form, the constraint $T \geq 0$ is added to the constraint store. The variable result is then bound to the value of the principal instance variable of the mortgage. Since the dotimes loop is just an abbreviation for a statement of the form

```
(let ((i 0))
  (loop
    (if (= i time)
        (exit-loop)
        <do the body of the loop and increment i>)))
```

we have to evaluate another if form. Since time is bound to a constraint variable, the expression (= i time) returns a constraint, and, as before, the if-operator evaluates both branches. To evaluate the first branch we add the constraint $T = 0$ to the constraint store; therefore evaluation proceeds by exiting the loop and returning the value of result to the caller of balance. At this point the variable x in the let*-form is bound to 100. In the next step we add the constraint $100 = 95$ to the constraint store and this path of the computation fails.

The program then proceeds to evaluate the alternative of the if-form. Incrementing the result variable and the loop counter i proceeds in the normal manner since all subcomputations return deterministic values. In the next iteration the above process is repeated with the constraint $T = 1$ added to the constraint store. In this case evaluation of the first branch exits the method with value 95; the constraint $x = 95$ is true, and the value 1 of T is returned. If we tried to search for further values the program would loop indefinitely.

This example demonstrates the facilities which allow the seamless integration of constraints into the functional core-language: First-class constraints (and constraint-variables), a facility for non-deterministic evaluation (backtracking) and polymorphic versions of the built-in functions which take constraints into account. With these additions to the language, functional programs can be used transparently in non-deterministic computations. The example also demonstrates that we have unfortunately introduced one of the problems that is common to all constraint languages into *ConS/Scheme*: It is easy to inadvertently write programs that run into infinite loops.

3 The Object-Functional Core

All facilities provided by *ConS/Scheme* are implemented in terms of a small set of powerful constructs. *ConS/Scheme* is based on the *Gauche* [7] dialect of Scheme [8], which offers a flexible object system with multiple inheritance and multiple dispatch. In the next subsections we give a short description of the syntax and some important features of Gauche. This introduction covers only a few topics and glosses over many important details. A more detailed description of object systems that are based on multimethods given, e.g., in [15].

3.1 The Base Language

Like most Lisp dialects, and unlike most other languages, *ConS/Scheme* has two representations for programs: Programs in *external representation* are sequences of characters in a file. Programs in *expression syntax* are represented by objects of the programming language; the most important aspect of this is that the program structure is represented by nested lists. If the first element of a list representing a program is one of a few special symbols (called *special operators)*, the form follows special evaluation rules and is called a *special form*. Macros can be used to rewrite list representations of programs. In [20] Steele and Sussman show that most programming constructs can be defined with macros and a small programming language core.

3.2 The Object System

The *Gauche* dialect of Scheme, on which *ConS/Scheme* is built, contains a powerful generic-function based object system. Classes define the structure of objects, instances of classes contain instance variables (called *slots*). By convention class names are enclosed in angle brackets, e.g., `<a-class>`, this has no semantic consequences.

In contrast to more conventional object-oriented languages a class (usually) contains no behavior in the form of methods, instead behavior is contained in generic functions. A *generic function* is a function whose behavior depends on properties of its arguments, most commonly the class of one or more arguments.

The behavior of a generic function is implemented by *methods*. A method definition in *ConS/Scheme* is similar to a method definition in an object oriented language. Since method definitions are not textually nested in class definitions and the *this* pointer has to be explicitly specified method definitions look syntactically similar to function definitions in functional programming languages.

4 Parametric Search

4.1 Search Strategies and Value Collectors

Some logic programming languages, e.g., *Prolog*, fix the order in which non-deterministic computations are evaluated. If this built-in search-strategy is not suitable for some application, the whole program has to be rewritten to implement the desired behavior. Therefore many modern logic programming languages allow a programmatic specification of the evaluation order in terms of a *search protocol*.

ConS/Scheme uses a search protocol to control the backtracking behavior. This protocol is based on the search strategies presented in [16]. In [16] the search strategies are used to search tree-like data-structures composed of *nodes*. In *ConS/Scheme* the applicability of the search protocol is extended to the control of the backtracking mechanism. A more detailed description of the search protocol is given in [4, 3].

Most logic programming languages provide the user with special predicates that control how many values of a computation should be computed and returned. In *ConS/Scheme* the user can define these predicates by implementing a *value collector* that controls the interface between non-deterministic computations and the surrounding deterministic context. Details can again be found in [4, 3].

5 Backtracking Search

The backtracking search mechanism consists of the single primitive `either-fun`, and two global parameters `current-search-strategy` and `current-collector`. A denotational semantics of *ConS/Scheme* is given in [4]. Intuitively, `either-fun` creates a *choice point*, and proceeds to evaluate one branch of the computation space. Which branch is initially taken, and where the computation is resumed after a branch fails or delivers a result is controlled by the two strategies. All other functions are implemented in terms of these primitives.

The function `either-fun` takes a variable number of nullary functions as arguments (these functions are also called *thunks*). It arranges for these functions to be called with the continuation which was current at the time `either-fun` was called. We call these combinations of functions and continuations the *nodes* of the *program tree*. Backtracking is invoked by either returning a value from a branch of the computation tree if the computation was successful or by calling the function `fail` to abort the current branch.

The macro `either` provides some syntactic sugar on top of `either-fun`:

```
  (either form ...)
```

is syntactically equivalent to

```
  (either-fun (lambda () form) ...))))
```

The macro `run` provides a mechanism to change the search strategy and value collector for some part of the computation.

As first example for the use of non-deterministic operations, consider the following program

```
(run ((all-values-collector) (depth-first-search))
  (either 1 2 (either ’a ’b ’c) 3))
```

The `either`-form defines a non-deterministic computation that branches into nodes evaluating to 1, 2, a nested non-deterministic computation and 3. The nested computation branches into three nodes evaluating to `a`, `b` and `c`. Evaluating this tree with an all-values-collector and depth-first-search returns the list `(1 2 a b c 3)`. If the program were changed to use breadth-first-search as search-strategy, the returned result would be `(1 2 3 a b c)`. The returned values can be chosen by changing the value-collector: if an instance of the class <one-value-collector> was used, the single value 1 would be returned (and the non-deterministic computation would not proceed after producing the first value), if an <all-symbol-collector> was used, the result would be (a b c), a <maximum-integer-collector> would return 3.

It is also possible to define non-deterministic functions. One simple example is the following:

```
(define (a-boolean)
  (either #t #f))
```

This function non-deterministically returns true and false. The values computed by a call to the function are again collected by a value collector and the result is returned as the result of the computation:

```
(if (a-boolean) 1 2)
```

This program returns the list (1 2). It is again possible to use different search-strategies and value collectors. Definitions of non-deterministic functions can also be recursive. The function

```
(define (an-integer-above m)
  (either m (an-integer-above (+ m 1))))
```

successively returns all integers larger than its argument.

Computations that cannot continue can be aborted by calling the function fail. This causes the current branch of the program tree to be removed without providing a value. Thus the program

```
(run ((all-values-collector) (depth-first-search))
 (either  1 (fail) 2 3 (fail)))
```

evaluates to (1 2 3). The following expression

```
(either (begin (display "one") 1)
        (begin (display "two") (fail) (display "still two") 2)
        (begin (display "three") 3))
```

prints "one" "two" "three" and returns (1 3). We can therefore define

```
(define (an-integer-between min max)
  (if (> min max)
      (fail)
    (either min (an-integer-between (+ 1 min) max))))
```

Since this function recurses in the either form, the evaluation tree is binary, with a value as left child and a non-deterministic computation as right child of all nodes but the rightmost one. To get a balanced tree another definition would be more appropriate, e.g.,

```
(define (an-integer-between min max)
  (cond ((< max min) (fail))
        ((= max min) min)
        (else (let ((result (quotient (+ min max) 2)))
           (either result
                   (an-integer-between min (- result 1))
                   (an-integer-between (+ result 1) max))))))
```

6 Integration of Constraints

With generic functions and non-deterministic computations at our disposal, the integration of constraints is relatively straightforward. Most special operators of the core language are defined as macros. In contrast to the expansions given in [20], most macros in *ConS/Scheme* expand into generic functions. This allows users of the language to extend the behavior of special operators. Therefore the constraint system can be defined on the "user level" of the language, without access to internals of the implementation.

To implement the constraint system we define classes `<constraint-variable>` and `<constraint>` and add appropriate methods to the already defined generic functions. The evaluation and binding operations of the base language remain unchanged, in particular it is not possible to bind constraint-variables to values. (It is, of course, possible to bind variables to constraint-variables and to constrain a constraint-variable to a single value).

Constraint-variables have a slot that describes the current set of values. If new constraints are propagated, the value of this slot is modified.

The generic definitions of most operators are straightforward. For example the function `+` may be defined as follows:

```
(define-method + ((x <number>) (y <number>))
  (%primitive-+ x y))
(define-method + ((x <constraint-variable>) (y <number>))
  (let ((result (make <constraint-variable>)))
    (add-constraint
     (make <sum-constraint> :lhs x :rhs y :result result))
    result))
(define-method + (x (y <constraint-variable>))
  (let ((result (make <constraint-variable>)))
    (add-constraint
     (make <sum-constraint> :lhs x :rhs y :result result))
    result))
```

In *ConS/Scheme* most control structures are defined in terms of a small set of primitive operations, like `lambda`-abstractions and function applications. There are no special operators that implement conditionals or control-flow manipulation, all these operations (except `call/cc`) are implemented in a library. To illustrate this point we show the definition of the "special operator" `if`:

We implement `if` as a macro that builds thunks for the consequent and alternative branches of the conditional and then calls a generic function `eval-if` to dispatch on the value of the predicate. Note that the body of the `eval-if` function *calls* the appropriate thunk to perform the evaluation of a single branch. If the predicate of a conditional evaluates to a constraint, both branches are evaluated non-deterministically by the third method.

```
(define-macro (if pred con . alt)
  ‘(eval-if ,pred
            (lambda () ,con)
            (lambda () ,(if (null? alt) #f (car alt)))))

(define-method eval-if (pred con alt)
  (con))
(define-method eval-if ((pred <false>) con alt)
  (alt))
(define-method eval-if ((pred <constraint>) con alt)
  (either (begin (add-constraint pred) (con))
          (begin (add-constraint (negate pred)) (alt))))
```

7 Interaction with Imperative Features

The integration of non-deterministic computation and the functional subset of *ConS/Scheme* results in a satisfying combination of functional and constraint-based programming. This is, however not always the case if imperative features of the language are used. The main problem is the following: If a branch of the computation tree performs a side-effect, should this effect be reversed if the branch terminates or not? It is evident that there is no “right” answer to this question.

This problem can be clearly seen in the implementation of the constraint solver: Each constraint variable contains a slot `domain` that holds information about the possible values under the current set of constraints. If new constraints are added, this slot is modified destructively. It is clear that all modifications made in a particular branch of the program tree have to be undone when the control flow leaves this branch. However, since the control flow of the program can be controlled by search and value strategies there is no simple stack discipline for performing these undo-operations. Furthermore, it is possible that a branch of the program tree is resumed after some of its side-effects have been undone; in this case the side effects have to be redone.

We currently have no automated solution for this problem. In the actual implementation of *ConS/Scheme* the `either-fun` function takes two additional thunks *before* and *after* as arguments. The *before* thunk is evaluated every time when the control flow enters the branch of the program tree dominated by `either-fun`, the *after* thunk is evaluated each time the control flow leaves this region. This solution allows the programmer to correctly manage the state even if search strategy repeatedly switch between different branches of the program tree. However a more elegant and less error-prone solution to this problem is desirable.

8 Conclusions and Further Work

We have shown that the combination of non-deterministic computation with a language based on generic-functions leads to a language that offers seamless integration of three important programming paradigms: object-oriented, functional and constraint-based. The resulting language *ConS/Scheme* offers simple operational and denotational semantics but provides the same expressive power as other constraint-functional languages. The prototype of *ConS/Scheme* is implemented on top of Gauche-Scheme.

There are some significant optimization opportunities that we would like to explore in the future: The most important bottleneck is the search protocol which relies heavily on generic function dispatch. In most programs the specific methods called could be determined statically and therefore most of the overhead of the protocol could be eliminated. It would be interesting to investigate possibilities to optimize the implementation of *ConS/Scheme* to eliminate most generic function calls [17].

We have already mentioned some unresolved problems when destructive modifications are performed in non-deterministic computations. While *ConS/Scheme* provides facilities that allow the programmer to write programs that work correctly when side-effects are performed in non-deterministic computations, this takes more effort than it should. The most likely approach to solve this problem seems to be by adding a transaction concept to the language.

Acknowledgments I would like to thank Professor Martin Wirsing for his continuing support. Parts of this work were developed while I was staying at Monash University, Melbourne, on invitation of Professor John N. Crossley.

Bibliography

[1] Thom Frühwirth, *Web page:* `http://www.pms.informatik.uni-muenchen.de/software/jack/index.html` *accessed 2. october 2003*.

[2] Michael Hanus, *Curry, an integrated functional logic language*, University of Kiel, Germany, 0.7.1 ed., June 2000, See also the URL `http://www.informatik.uni-kiel.de/~curry/`, accessed 29 October 2001.

[3] Matthias M. Hölzl, *ConS/Lisp—a mop-based non-deterministic lisp*, Proceedings of the International Lisp Conference (Raymond de Lacaze, ed.), 2002.

[4] ______, *Constraint-functional programming based on generic functions*, Workshop Proceedings: MultiCPL'02: Workshop on Multiparadigm Constraint Programming Languages (Michael Hanus, Petra Hofstedt, Slim Abdennadher, Thom Frühwirth, and Armin Wolf, eds.), September 2002.

[5] ILOG, *Web page for ILOG solver:* `http://www.ilog.com/products/solver/` *accessed 29 october 2001*.

[6] ______, *Web page for the ILOG optimization suite:* `http://www.ilog.com/products/optimization/` *accessed 29 october 2001*.

[7] Shiro Kawai, *Web page for gauche:* `http://www.shiro.dreamhost.com/scheme/gauche/`, *accessed 2. october 2003*.

[8] R. Kelsey, W. Clinger, and J. Rees, *Revised^5 report on the algorithmic language scheme*, Higher-Order and Symbolic Computation **11** (1998), no. 1.

[9] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow, *The art of the metaobject protocol*, The MIT Press, 1991.

[10] Rita Loogen, *Integration funktionaler und logischer programmiersprachen*, Oldenburg, 1995.

[11] Kim Marriott and Peter J. Stuckey, *Programming with constraints — an introduction*, The MIT Press, 1998.

[12] David McAllester, *Lifting*, November 1992, Lecture Notes for 6.824, Artificial Intelligence.

[13] David McAllester and Jeffrey Mark Siskind, *Nondeterministic Lisp as a substrate for constraint logic programming*, Proceedings AAAI, July 1993, pp. 133–138.

[14] ______, *Screamer: A portable efficient implementation of nondeterministic Common Lisp*, Tech. report, University of Pennsylvania, Institute for Research in Cognitive Science, 1993, IRCS-93-03.

[15] Kent M. Pitman (ed.), *Information technology — programming language — common lisp*, ANSI Standard, no. ANSI X3.226–1994, American National Standards Institute, 1994.

[16] Stuart Russel and Peter Norvig, *Artificial intelligence, a modern approach*, Prentice Hall, 1995.

[17] Olin Shivers, *Control-flow analysis of higher-order languages*, Ph.D. thesis, Carnegie Mellon University, May 1991, CMU-CS-91-145.

[18] Gert Smolka, *Web page for the* Alice *programming language:* `http://www.ps.uni-sb.de/alice/`, *accessed 29 october 2001*.

[19] ______, *Concurrent constraint programming based on functional programming*, Proceeding of the European Joint Conferences on Theory and Practice of Software (ETAPS), 1998.

[20] Guy Lewis Steele Jr. and Gerald Jay Sussman, *Lambda the ultimate imperative*, Tech. Report AI Memo 353, Maddachusetts Institute of Technology, Artificial Intelligence Laboratory, March 1976.

Functional versus OO Programming: Conflict Without a Cause

DeLesley Hutchins

CISA, School of Informatics, University of Edinburgh
Appleton Tower, 11 Crichton Street,
Edinburgh, EH8 9LE, UK
+44 (0) 131 650 2732
D.S.Hutchins@sms.ed.ac.uk

Abstract. Despite some overlap, object-oriented and functional methodologies are generally regarded as separate paradigms. OO programming is concerned with classes and inheritance, whereas functional programming is concerned with functions and function composition. I present a new object model which unifies these paradigms. Classes are represented as higher-order functions, and inheritance becomes a form of function currying. This unification results in a far simpler and more flexible object model, and one which eliminates several outstanding problems with existing OO languages.

1 Introduction

Although they are generally presented as separate programming paradigms, functional and object-oriented languages are closely intertwined. Each borrows concepts from the other, to the extent that it is almost impossible to describe any language as being either "pure functional" or "pure object-oriented". This mingling can be most clearly seen in mainstream OO languages: classes and interfaces are built from methods, and methods are simply functions. Likewise, even the purest of functional languages, such as Haskell or ML, use OO concepts. Functional programs must still deal with data types, and a combination of subtyping (a.k.a. inheritance) and function overloading is sufficient to mimick most OO constructs. Haskell provides a particularly powerful mechanism for dealing with classes and polymorphism that rivals any mainstream OO language on the market. [8]

Nevertheless, this kind of intertwining can best be described as "peaceful coexistence," rather than a more meaningful marriage. Little attempt has been made to unify the functional and OO paradigms into a single whole. In C++, for example, methods (i.e. functions) and classes are regarded as completely different constructs. Classes can inherit from one another, but functions can not. Classes have constructors and destructors, public and private methods, and they can be nested, all properties that C++ functions do not have. Methods, on the other hand, can be declared **virtual**, and function pointers can be passed as first-class objects at run-time. C++ does not support virtual classes or class meta-objects.

Haskell has a similar set of limitations. Indeed, classes and functions are so different from one another that there seems little point in unifying them. Classes are types;

their purpose is to label data structures with meta-information so that the code which manipulates that data "knows" what kind of data it is dealing with. Inheritance creates a subtype relationship; it organizes a set of related classes into categories, so that similar types of data can be handled in a consistent way.

Functions serve a very different purpose. A function encapsulates a computation, not a type. A function manipulates data; it does not label it or categorize it. Even OO languages obey these principles – an instance of a class stores data, while methods (a.k.a. functions) modify and interpret it.

Despite these obvious differences, there is a deeper symmetry between classes and functions which has been largely ignored. Although they serve different purposes, the computational mechanism by which classes and functions are implemented is very similar. An instance of a class and the activation record of a function are almost exactly the same. When a class is instantiated, a new record is created in which the data members (the slots) of the class are bound to actual values. The act of calling a function likewise creates an activation record in which the arguments of the function are bound to values. Both instances and activation records create a lexical context in which objects can be referenced by name.

There are only two real differences between calling a function and instantiating a class. A function has a function body, and it returns a value. This capability can be easily implemented with class constructors. For example, here is the factorial function implemented as a C++ class:

```
class Factorial {
 public:
   int n, result;
   Factorial(int n_) : n(n_) {
     if (n <= 1) result = 1;
     else result = n*Factorial(n-1).result;
   }
};
Factorial(3).result;   // returns 6
```

This symmetry extends to other constructs as well. The factorial function can also be implemented with C++ templates using a similar technique:

```
template <int N>
class Factorial {
 public:
   enum { result = N * factorial<N-1>::result };
};

class Factorial<1> {
 public:
   enum { result = 1 };
};
```

Both the class and template implementations create a return value by storing it as a data member of the class. The class definition is a bit more straightforward, because a function body can be emulated simply by placing code in the constructor. The template must jump through a few extra hoops; it declares a compile-time expression with an **enum**, and uses template specialization because C++ does not provide a compile-time **if** statement. For the most part, however, these are merely implementation details.

In other words, C++ provides three completely different constructs: functions, classes, and templates, which in this case do essentially the same thing. All three constructs create a lexical scope in which named parameters can be bound to values. (I will use the terms parameters, slots, arguments, methods, and members somewhat interchangeably throughout this paper. I believe "parameters" to be the most general term; the others have more specific connotations in various languages.) Since the lambda calculus already provides a mathematical formalism for binding named parameters, this raises an obvious question: can the functional and object-oriented paradigms be unified?

The answer to this question is that not only can the two paradigms be unified, but that such unification has considerable expressive power. For example, in most functional and OO languages, only methods (i.e. functions) can be declared **virtual** and overridden during inheritance. In C++, this is because only function pointers can be stored in the virtual method table. In Haskell, it is because only functions can be overloaded to handle polymorphic types. Yet the notion of a "virtual class" actually turns out to be quite useful. Virtual classes not only provide an elegant mechanism for implementing generic classes (i.e. C++ templates), they are a powerful tool for large-scale programming. [13]

A single class encapsulates a set of interacting methods together with the data they operate on, and allows the whole set to be extended, modified and reused by means of inheritance. Unfortunately, most complex problems require more than once class. With virtual classes, a set of interacting classes can likewise be encapsulated, and then extended, modified and reused by means of inheritance. Virtual classes thus provide much of the capability offered by components or aspect-oriented programming, [1] [12] without any need to abandon standard OO concepts. I will discuss this and other issues later on in this paper.

I will present my ideas for synthesis in the form of the Ohmu programming language, an experimental language which is being developed at MZA Associates Corporation.[1] The Ohmu language provides only one construct: the structure, and one operation: the structure transformation. Functions, classes, and templates, along with function calls, inheritance, and instantiation, can all be emulated with structures and structure transformations.

This paper is organized as follows. Section 2 introduces the Ohmu object model, which unifies functions and classes. Section 3 explores issues related to binding and types. It explains why existing type systems are inadequate, and introduces the Ohmu prototype model, which eliminates the distinction between types and values. Section 3 also describes the similarity between function currying and OO inheritance. Section 4 discusses lazy evaluation, meta-programming, and issues related to run-time and

[1] This work is being funded by:
MZA Associates Corporation
2021 Girard SE, Suite 150, Albuquerque, NM 87106-3140
voice: (505) 245-9970 fax: (505) 245-9971 web: http://www.mza.com

compile-time. The Ohmu language depends on a partial evaluation engine to shift code from run-time to compile-time, and track dependencies which are introduced by the use of virtual types. Section 5 concludes with a discussion of how these concepts relate to large-scale programming.

2 Synthesis

As I mentioned earlier, the most fundamental difference between classes and functions is the fact that classes represent types, whereas functions do not. Every object must be tagged with an identifying type, namely the class that created it, and the compiler uses that information to make sure that programs are type-safe. Types may encode varying amounts of information depending on the language, but in general a type describes the structure of data within an object, and the valid operations that can be applied to that data. In other words, the type or class of an object describes the interface of that object.

Another important characteristic of classes is that they exist at the meta-level of program specification. A class describes the general properties of a set of similar objects, rather than the specific details of one particular object. Classes and types thus exist at a higher level of abstraction than "real" objects and executable code. In most languages, including Java, C++, and Haskell, types are not first-class objects, and they exist only at compile-time.

The traditional view holds that while human programmers and compilers must both be aware of types in order to reason about program correctness, a running program should only be concerned with "real" objects. Program fragments that do need to reason about types are known as meta-programs, and they can be difficult to write because they must essentially extend the operation of the compiler at compile time. C++ templates are perhaps the most notorious example of the difficulties involved with meta-programming. [5]

Functions are much simpler than classes. A function has a type just like any other object, but it is not a type in and of itself. There is no such thing as an instance of a function. Functions do exist at run-time, and they are treated as first-class values in most languages, even in OO and imperative languages like C++. The purpose of a function is to encapsulate a particular computation into a reusable module.

Mathematically, a function is defined as a mapping from one set to another:

$$\begin{aligned} \text{factorial} : \quad & \mathbb{Z} \rightarrow \mathbb{Z} \\ \text{add} : \quad & \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \end{aligned}$$

The *factorial* function maps from the set of integers to the set of integers, while the *add* function maps from a pair of integers to a single integer. Note that $\mathbb{Z} \rightarrow \mathbb{Z}$ is a type: it describes the set of all functions that map integers to integers, while *factorial* itself is an instance: a single function within that set.

There are many ways to define a type mathematically, but the simplest is to state that a type is just a set — it is the set of all objects that have that type. There's $\mathbb{Z}$, the set of integers, $\mathbb{R}$, the set of floating point numbers, etc. More complex data structures can be represented as the product of one or more simpler types, e.g.

$$\begin{aligned} \mathrm{Point} &= \mathbb{R} \times \mathbb{R} \\ \mathrm{Line} &= \mathbf{Point} \times \mathbf{Point} \end{aligned}$$

If we define types as sets, however, then a function actually is a type. A function can be represented as a relation between its domain and its range. A functional relation is a subset of the product of the domain and the range, which satisfies the following criteria: for every element x in the domain, there exists one and only one element (x, r) in the relation. The factorial function can thus be defined as:

$$\mathrm{factorial} = \{(0,1),(1,1),(2,2),(3,6),(4,24)\ldots(n,n!)\} \subset \mathbb{Z} \times \mathbb{Z}$$

According to this model, we can now define what it means to instantiate a function. An instance of a function is an activation record for that function – a data structure that contains both the bound arguments to the function, and the result of evaluating the function with those arguments. This is exactly what the C++ class version of *factorial* defines: it's a class with two elements, n and result, and the constructor initializes result to the appropriate value.

Mathematically speaking, there is no reason to distinguish between functions and classes. Functions and classes are both types, and they can both be instantiated. This property has not been exploited in other languages, most likely because it was not considered particularly useful. When a function is evaluated, we are generally only interested in the result, and the activation record is discarded immediately. Since activation records do not persist, they need not have a type. C++, for instance, allocates all activation records on the stack, which means that the record has already been destroyed by the time the function returns. Lisp and similar languages may keep such records around as a lexical context for closures, but activation records are still not first-class values and thus have no type.

If activation records are allowed to persist, however, then they can serve as instances, and functions can consequently serve as classes. There is no need for a functional language like haskell to define a separate mechanism for declaring data types. From the opposite perspective, there is no need for an OO language like Java to define a separate mechanism for declaring functions; classes can be used just as effectively.

The Ohmu language emualtes both classes and functions by with the Ohmu *structure*:

```
factorial: Struct {               // a function
  n:        Integer;
  result: if (n == 0) then 1 else n*factorial(n-1);
  new:                bind(n);
  implicit call:  bind(n) = result;
};

Point: Struct {                    // a class
  x,y:    Float;
  r:      sqrt(x*x + y*y);         // -- a method
```

```
    theta:  atan2(y,x);              // -- another method
    new:    bind(x,y);
  };

  x:      factorial.new(3).result; // x = 6 sugar-free
  y:      factorial(3);            // y = 6 with sugar
  origin: Point.new(0,0);          // instantiate a class
```

Here is the Ohmu definition for both a class and a function. The **bind** command is responsible for binding values to parameters; it essentially acts as a named constructor, and it is declared with an ordered list of the names that it needs to bind. Unlike a traditional function, an Ohmu structure seldom binds all of its parameters. Some members, such as r, theta, and result, are internal methods and do not need to be bound. Structure parameters are also referenced by name, so they have no set order.

The **bind** command thus specifies which parameters should be bound, and what the order of arguments should be for binding them. It creates and returns an instance of the structure that it is declared in. Instantiation in Ohmu is called a *structure transformation*, because it has somewhat different semantics than instantiation in other OO languages.

A **bind** declaration may optionally specify a result. This is a bit of syntactic sugar; it keeps the programmer from having to put a .result after every function call. The **implicit** keyword is another bit of sugar; it operates like **operator()** in C++, and eliminates the need to write out an explicit .new. Note that new here is not a keyword. Ohmu supports named constructors, and while new is generally used by convention, other names, such as call in the example above, are perfectly legal.

This unification between classes and functions is not unique to Ohmu. It was originally developed as part of the Beta language. [14] In Beta, both classes and functions are referred to as *patterns*. Each pattern has an optional body which contains statements that are evaluated when the pattern is instantiated. Ohmu structures differ somewhat in that there is no real function body. Ohmu functions are instead treated as relations, and the code which computes the result is stored as a named member of the structure. Incidentally, the Point class above is also a relation — it calculates r and theta for every x and y. As a result, Point can be used either as a class, or as a function that converts from Cartesian to polar coordinates.

It should be noted that although Ohmu functions are really structures, the syntax given above can be a bit unwieldy. The Ohmu standard library includes a **Function** macro that automates this process:

```
  factorial: Function((n: Integer), Integer) {
    if (n == 0) then return 1
    else return n*factorial(n-1);
  };
```

This alternate syntax uses a more conventional statement list for the function body, but it is otherwise equivalent to the earlier definition.

2.1 Single vs. Multiple Dispatch

Another major difference between OO and functional languages is the way they implement methods. In an OO language, methods are usually visualized as being part of the object. They not only manipulate object data, they provide a high-level interface that controls access to that data and guarantees object integrity. OO methods are *overridden* with new versions during inheritance in a way that closely resembles parameter binding.

In a functional language like Haskell or CLOS, the methods defined on a data type are declared as external functions, and different versions of a method are defined for different derived types by means of function *overloading*. The advantage of this system is that it supports multiple dispatch, a technique in which the types of all arguments to a function are considered when determining which overloaded version to call. Most OO languages rely on single dispatch, which only considers the type of the message receiver.

The problems with single dispatch are well documented: it can be difficult to define certain polymorphic operations for a set of related types using single dispatch. [4] A classic example would be arithmetic operators such as +, $*$ and the like. Such operators must be able to handle integers, floats, complex numbers, etc., and the types of all arguments should ideally be considered when determining which version of + to call. Multiple dispatch also makes it easier to add new operations to existing data types, because the addition of new methods does not affect the class hierarchy.

The disadvantage of defining methods outside of the data type is that object data and the functions that manipulate that data are not encapsulated into one unit. There is no longer a clear interface to a class other than the slots of the data type itself, nor is it obvious which methods need be overloaded to support new derived types. Single dispatch makes it easier to define new classes with existing operations, whereas multiple dispatch makes it easier to define new operations on existing classes. Single dispatch also gives an object much tighter control over its own integrity, since it is always obvious which method will handle a particular message.

2.2 Higher Order Functions

The Ohmu language uses single dispatch for a different reason. Single dispatch allows inheritance and instantiation to be unified into a single operation. OO languages have traditionally used two different kinds of binding: instantiation binds data members to values, while method overriding during inheritance binds new definitions to existing methods. Even the Beta language maintains this distinction. Beta demonstrated that functions and classes could be unified into a single construct, but it still provides two operations; data members are bound to create instances, while methods and types are bound to create derived classes.

The Ohmu language goes one step further. If methods are defined inside a class then they become parameters of that class. If functions and classes are also two sides of the same coin, then a class that declares internal methods is analogous to a higher order function. A higher-order function is one that accepts other functions as arguments, and/or returns another function as its result. By including other functions and classes as internal parameters, a class becomes a higher-order construct.

Functional languages have long supported higher-order functions; indeed, they are one of the most powerful mechanisms provided by the functional programming paradigm. [8] Higher-order functions rely on the principle that functions are first-class values which can be passed around at run-time just like any other object. Passing one function as an argument to another is no different from passing any other value.

OO languages provide a special operation that only binds functions — namely inheritance. Yet if functions are first-class values, then there is no need for a separate inheritance operation just to override methods. In fact, simple instantiation is enough to implement virtual methods, even in C++:

```
class Number {
 protected:
   // store the add method as a function pointer
   typedef void (*operation)(Number& self, Number& n);
   operation add;

 public:
   // decent syntax for calling add
   inline void operator+=(Number& n) {
     (*add)(*this, n);
   }

   Number(operation a) : add(a) { }
};

class Integer : public Number {
 private:
   int value;
   static void add_op(Integer& self, Integer& n) {
     self.value += n.value;
   }

 public:
   Integer() :  // initialize the add member
     Number(add_op), value(0)
   { }
};
```

The code above is a toy implementation of a Number and Integer class. Instead of using normal virtual methods, Number creates its own virtual method table by storing pointers to functions in the object itself. Its behavior is equivalent to that of standard virtual methods.

2.3 Higher Order Functions = Reusable Code

Stepping back a little, I wish to discuss briefly why both classes and higher-order functions matter at all. Functions in general provide a convenient abstraction for performing computations because the implementation of a function is completely hidden behind a clear and well-defined interface. Mathematically, a function is just a relation between two sets. It does not matter how a function arrives at a result, it matters only that it takes values of one type, and maps them onto values of another type.

The downside is that because the implementation is completely opaque, there is no way to modify or re-use parts of a function definition. Object-oriented languages address this problem by means of block structure and inheritance. Unlike functions, classes are not monolithic objects; a class consists of a group of interacting methods which can be selected and overridden individually in derived classes. Virtual methods act as *hooks*; they provide named locations where a base class can defer certain parts of its implementation to its children, or where new functionality can be inserted into an otherwise working system.

Functional languages provide an analogous mechanism for code reuse in the form of higher-order functions. A higher-order function, like a class, defers part of its implementation to hooks – other functions that are passed as arguments. By attaching different operations to the hooks, a whole family of related high-level functions can be produced, just as a whole family of derived classes can be created by overriding the virtual methods of a base class.

The classic example from functional languages is the map routine, shown here as it is defined in Haskell:

```
map f []    = []
map f (h:t) = (f h) : (map f t) -- (h:t) = (head:tail)

map (*2) [0,1,2,3,4]             -- returns [0,2,4,6,8]
```

The map routine takes a function f and a list (h:t) as arguments, and applies f to every element of the list, thus creating a new list. The OO equivalent of map is the iterator design pattern, [7] which likewise hides the details of data structure traversal:

```
List::iterator it = myList.begin();
List           newlist;
for (;it != myList.end(); it++)
  newlist.push_back(it.value() * 2);
```

Both classes and higher-order functions share a common purpose. They structure code in such a way that a general algorithm, in this case a data structure traversal, can be glued together with third-party code, in this case an operation to be performed on each element. What makes a language powerful is not the number or complexity of its constructs, but the "glue" that allows different constructs to be combined. The Ohmu language is an attempt to provide a better glue by treating all constructs in a uniform manner.

2.4 Higher-Order Classes, a.k.a. Generics

Since Ohmu functions and classes also represent types, the analogue of higher-order functions is higher-order types. A higher-order type is merely a type that is parameterized by other types. In OO parlance, such types are referred to as *generic classes*.

Generic classes in C++ are implemented with templates:

```
template<class T>
class List {
 public:
   T        head;
   List<T>* tail;
};

// creates a new class -- lists of integers
typedef List<int> IntegerList;
```

A C++ template can be viewed in several ways. On the one hand, a template acts as a compile-time function that maps from types (or constants) to classes. On the other hand, a template represents a type in and of itself. More accurately, templates *should* represent types; it would be extremely helpful if the definition above declared a generic type List, and if IntegerList was a specific instance of that type. That's how Java generics work. [3] Unfortunately, a C++ template cannot be used by itself; it must be instantiated before it means anything at all. This was a major oversight in the design of C++; templates are an excellent example of why it is useful for functions and types to have a unified definition.

3 Prototypes

I have now discussed three different ways in which parameters can be bound. Data members are bound to values during function calls or class instantiation. Methods are normally overridden during inheritance, but they can also be bound during instantiation so long as functions are treated as first-class objects. Type parameters are bound during template instantiation. (It should be noted that type parameters are also bound at compile-time rather than run-time, but I will ignore that for the time being; binding times are discussed in Sec. 4.)

The astute reader will have noticed that I have not yet given any examples of type and method binding in the Ohmu language. This is because the semantics of binding data, functions, and types are all slightly different. C++ solves this problem by providing three different operations: instantiation, inheritance, and template instantiation, but the Ohmu language only provides one operation, and that means that the semantics must be unified.

When data parameters are bound, the binding moves from an abstract type declaration, such as **int** or **float**, to a concrete value, such as 7 or 3.14. A traditional second-order function parameter does the same thing. It is declared as an abstract function type such as $\mathbb{Z} \rightarrow \mathbb{Z}$, which represents a set of possible functions, and it is then bound to a concrete function definition such as factorial .

OO method overriding, on the other hand, may start out with a method that already has a concrete definition, in which case the method is *replaced* with another that has the same type. The semantics of object replacement are different from the semantics of binding. Object replacement moves from value to value, whereas binding moves from type to value.

The situation becomes even more confusing once we introduce template or generic class parameters. A constrained class parameter, such as those provided by Java generics, [3] specifies a base class. Such a parameter can be bound to any derived class of that base class. This, too, is different from data members. If a data member specifies a class, it means that the data member can be bound to any *instance* of that class, or any *instance* of a derived class. When a type parameter specifies a class, it means that the parameter can be bound to derived *classes*, but not instances.

To summarize, the three forms that parameter binding can take are:

- Type to Value: e.g. **int** ⇒ 3
  or **int** foo() = 0 ⇒ **int** foo() { . . . }
- Value to Value: e.g. **int** foo() { *def #1* } ⇒ **int** foo() { *def #2* }
- Type to Type: e.g. **class** Animal ⇒ **class** Dog

Another factor to consider is the fact that while both inheritance and instantiation can be implemented with binding, they are used in different ways. Instantiation is a “one shot” operation. It takes an abstract type declaration and creates an instance from it. Once an instance has been created, everything in that instance is fully bound, and definitions can no longer be re-bound.

Inheritance, on the other hand, is incremental. There may be a chain of derived classes, each of which re-binds various methods. GUI toolkits often have quite deep inheritance hierarchies, such as:

Object ⇒ Window ⇒ Widget ⇒ Control ⇒ Button

Each class in the chain will generally override key methods such as draw() or handleEvent() in order to implement its behavior.

3.1 Prototypes Unify Classes and Instances

One way to allow incremental changes in a uniform manner is to cease distinguishing between abstract types and concrete instances, and to define binding in terms of object replacement. If types are regarded as first-class objects, then this is a logical next step. Binding an abstract type to a concrete value becomes a simple matter of replacing one object: the type, with another object: the value. It does not matter whether the objects in questions are “types” or “instances”; we are simply replacing one object with a different, yet compatible object.

Unfortunately, this confounds the traditional mechanism for determining which objects are “compatible,” i.e. the type system. A parameter can only be bound to an object of the proper type; to do otherwise would violate the interface of the function or class. If we treat parameters as ordinary objects, and “bind” them by performing object replacement, then we must replace the original parameter with one of the same type. This is how standard OO inheritance works; a method can only be overridden with another method that has the same signature.

In most current OO languages that support first-class types, such as Smalltalk or CLOS, class meta-objects like **int** or **float** are instances of type Class. [11] The number 3 is not a class, and thus cannot be used to replace a parameter that has been declared as **int**. The object 3 and the meta-object **int** are not type-compatible. Smalltalk supports class meta-objects, but it still distinguishes between classes and instances, and object replacement is not an appropriate operation.

Consider also the case in which a parameter from a generic class must be specialized to a derived type, such as from **class** Animal to **class** Dog. If we used simple object replacement, **class** Animal could be replaced with any other class, even completely unrelated classes, because it is an instance of type Class. Generics require a more specific kind of parameter binding; the replacement object must be a derived class of the original.

The Ohmu language resolves these issues by switching from a traditional class/instance or type/value system to a prototype model. In the Ohmu prototype model, not only are all types first-class objects, but all objects are first-class types. As I discussed in section 2, a type identifies a set of objects. The type **int**, for example, identifies the set of all 32-bit integers. The number 3 can also be regarded as a set. It is a set with one element, otherwise known as a singleton set, which identifies the set of all 32-bit integers equal to 3. By granting ordinary objects the status of types, we can derive an appropriate replacement rule for parameter binding:

- An object can be replaced with any other object, so long as the new object is a subtype of the original.

Thus, we can replace the **int** object with the 3 object, because 3 is a subset of **int**. We can also replace **class** Animal with **class** Dog, so long as Dog derives from Animal. This new rule for parameter binding encompasses all of the traditional OO and functional operations. It handles function calls and class instantiation by binding types to values, it covers OO inheritance by allowing method overriding, and it allows template and generic class parameters to be refined to derived classes.

3.2 Computations with Prototypes

Treating values as types has some interesting consequences. First of all, it means that values and types must have the exact same interface. In other words, if it is possible to evaluate an expression like (3 + 1), it must also be possible to evaluate the expression (Integer + 1). If this were not true, then replacing Integer with 3 would not be a type-safe operation.

So what does the expression (Integer + 1) mean? An abstract type such as Integer represents a “don’t know” value. We don’t know what the result of adding 1 to an arbitrary integer is, but we do know that the result will be an integer. So Integer + 1 = Integer.

A side benefit of this system is that it becomes easier to write meta-programs that reason about types. The type of an expression can be determined merely by evaluating an expression with abstract prototypes. Consider the Point class from before:

```
Point: Struct {
  x,y:    Float;
  r:      sqrt(x∗x + y∗y);
  theta:  atan2(y,x);
  new:    bind(x,y);
};

point11: Point.new(1,1);

Point.r          // evaluates to sqrt(Float) = Complex
Point.theta      // evaluates to Float
point11.r;       // evaluates to sqrt(2) = 1.414...
point11.theta;   // evaluates to pi/2    = 1.570...
```

The Point prototype is a full-fledged object. It is perfectly legal to call methods such as r and theta on it; the compiler will return a prototype that represents as much as it is able to determine about the result. Once x and y have been bound to more specific values, r and theta will compute a more specific result.

Programming with prototypes has a somewhat different feel than programming with types and values. The word "prototype" is appropriate. On the one hand, prototypes represent abstract concepts. On the other hand, prototypes are real, working objects, and they can be used in real computations. Unlike traditional types and classes, prototypes are not meta-level constructs.

Traditional OO classes are descriptions of instances, and type-checking or other program verification is done by analyzing the class description, without running any user-level code. An Ohmu program is constructed differently. An Ohmu prototype defines a new concept not by describing it, but by building a working version of it. That working version can then be *specialized* by replacing abstract parameters with more specific subtypes.

Although I have thus far used the term "binding" to describe this process, "specialization" is actually a better word. Ohmu prototype specialization bears more resemblance to object-oriented inheritance than it does to traditional argument binding in functions. In particular, specialization can be incremental; it is possible to create a series of derived prototypes, each of which is more specific than its parent.

The simplest example of incremental specialization happens when some parameters are bound to concrete values, while the rest remain abstract:

```
XPoint: Point.new(Float, 0);    // a point on the x-axis
YPoint: Point.new(0, Float);    // a point on the y-axis

origin: XPoint.new(0, 0);       // binds both x and y
```

In XPoint above, y is bound to 0, while x remains an abstract prototype. XPoint thus represents an abstract type — the set of all points on the x-axis.

(In theory, a clever compiler might be able to determine at this point that $sqrt(x^2) = |x|$, and that $atan2(0, x) = \{0, pi\}$, and update r and theta appropriately. In reality, the current implementation has no symbolic math processing capability to speak of. Due to compiler limitations, the result of evaluating an expression with abstract prototypes may not be the most mathematically specific type possible. Instead, it represents what the compiler can guess about that expression. This is arguably a more useful result, since the capabilities of the compiler place a fundamental limitation on the rest of the code. The worst case scenario is that the compiler simply returns Object — the most generic type possible.)

Incremental specialization also occurs when parameters are bound and re-bound to a succession of abstract subtypes. A parameter declared as an Object, for example, could be bound to Number and then re-bound to Integer before finally being fully specialized down to 0:

```
List: Struct {
  head: Object;
  tail: List;
  new:  bind(head, tail);
};

// an abstract type -- a list of integers
IntegerList: List.new(Integer, IntegerList);

// an infinite list of zeros
ZeroList: IntegerList.new(0, ZeroList);

// a "normal" list -- (0,1,2)
finiteList: List.new(0, List.new(1, List.new(2, nil)));
```

In this example, List defines an abstract data type: a list of objects. IntegerList specializes head to create a new derived type: a list of integers. IntegerList is a subtype of List, just as it should be. ZeroList, in turn, is a subtype of IntegerList. It represents a type too: an infinite list of zeros.

Ohmu does not require any run-time storage to handle such an infinite list of zeros; the definition of ZeroList is encoded directly into the type system at compile-time. ZeroList is an example of a *lazy data structure*, a construct found in several functional languages, including Haskell. Such structures are often useful for representing abstract concepts in a convenient way. Here's a more complex example – the list of all natural numbers:

```
NaturalList: {
  head: Integer;
  tail: new(head+1);    // lazy parameter
  new:  bind(head);
};

NaturalNumbers: NaturalList.new(0);
```

By encapsulating the set of natural numbers as a list, list processing algorithms can traverse it using the same interface as that for "normal" lists.

3.3 Inheritance

Points and lists are simple classes with only a few data members, so the **bind** syntax that I have used up until this point is reasonably convenient. With large classes, however, it is not so convenient, for one simple reason. The **bind** command emulates a functional syntax — it accepts an ordered list of arguments, and maps each position in the list to a name in the structure. The expression **bind**(x,y,z), for instance, binds the first argument to x, the second to y, and so on.

A functional syntax is appropriate when the number of parameters is small, because it is simpler to list things in order than it is to specify them by name. When the number of parameters grows large, however, passing around long lists of arguments is both difficult and error-prone. The extreme example would be method overriding during inheritance; trying to create a derived class by passing it an ordered list of all methods would be an interface disaster.

Moreover, creating a derived class usually involves overriding only some of the definitions in the class, while leaving the others unchanged. For this reason, Ohmu supports an alternate inheritance-like syntax in which parameters can be specialized by name:

```
XPoint: transform Point { y: 0; };
```

This is an alternative way to declare the XPoint prototype that I described earlier. Only y needs to be specialized; x remains unchanged. The semantics of **transform** is identical to **bind**; it differs only in syntax.

3.4 Function Currying

This form of binding resembles another feature commonly found in functional languages: *function currying*. Function currying is a technique wherein a function with multiple arguments can be logically represented as a higher-order function with only a single argument. Ordinarily, a function with n arguments binds all of its arguments at once, and then returns a result. A curried function binds its first argument, and then returns another function of $n - 1$ arguments. It does not compute its “real” result until all arguments have been bound.

Ohmu structure transformations operate in a similar manner. Binding one parameter to a more specialized definition simply returns another structure. There are two main differences between function currying and structure transformations. The first is that transforming a structure with n parameters will return another structure with n parameters, not $n - 1$ parameters. This difference is due to the fact that the parameters in an Ohmu structure can be re-bound multiple times, so there is not necessarily any well-defined point in the life of a structure when all parameters can be said to be “fully bound”.

The second difference is that structure transformations can bind parameters out of order and by name, whereas a curried function can only bind arguments in the order in which they are declared.

3.5 More Inheritance

Even this modified definition of function currying is not a complete description of OO inheritance. I have spent a great deal of time up until this point discussing *binding*, because that is the area where the various OO and functional constructs differ the most.

Nevertheless, OO inheritance involves more than just method overriding; it also allows new methods to be added to a structure. Inheritance is a combination of two operations – aggregation and specialization. The **bind** and **transform** keywords only implement specialization. They can be used to modify existing parameters, but they cannot be used to add new ones, and they are thus insufficient to fully implement inheritance.

The Ohmu language, however, already supports aggregation. The very act of declaring a new structure groups a set of simpler objects into a compound aggregate. Object oriented inheritance can thus be emulated by combining the two operations. A true derived class is created by first transforming the base class, and then embedding it within a new structure. The **extends** keyword, which is syntactically the same as **transform**, will set up an appropriate embedding:

```
Point: Struct {
  x,y: Number;
};

Pixel: Struct {        // create new structure
  extends Point {      // transform base class
    x,y: Integer;      // specialize x and y
  }
  color: Integer;      // add a color parameter
};
```

The **extends** keyword will also do something that **transform** does not do — it deals with multiple inheritance in an appropriate manner. Like CLOS, Ohmu supports multiple inheritance by *linearizing* the inheritance tree. Linearization solves the dreaded “diamond” problem of multiple inheritance by transforming a multiple inheritance graph into a single inheritance tree.

Linearization is a good way to implement so-called *mixin* classes. [2] Mixins are a group of derived classes that all inherit from a common base class. Each mixin transforms the base class in a certain way in order to add a particular feature. These features can then be composed together by using multiple inheritance. [6] The linearization algorithm will order a set of mixin classes into a stack of software layers, where each layer modifies the layers beneath it. [16] The same “diamond pattern” that is regarded as a flaw in inheritance by Java and C++ then becomes a powerful tool for feature composition. [1]

The Ohmu implementation of linearization differs from CLOS only in that it can be used with more than just methods. Any parameter can be specialized during an Ohmu structure transformation, and linearization will re-order all such transformations.

3.6 Generic Classes and Virtual Types

Prototype specialization is general enough that it can do more than simply emulate standard OO inheritance; it can also emulate virtual types and generic classes. [17] The List class in section 3.2 is one example of a generic class. It differs from generic classes in other languages because there is no explicit type parameter (e.g. List<T>) as would be required in C++ or Java. Instead, the head parameter is specialized directly to a subtype.

Explicit type parameters are still useful, however, when there are several parameters of a structure which must have matching types. In a complex number class, for example, the real and imaginary parts should have the same type:

```
Complex: Struct {                    // a ”generic class”
  NumType:     Number;               // type parameter
  real, imag:  NumType;
  of:          bind(NumType);
  new:         bind(real,imag);
};

// this redefines real and imag
ComplexFloat:  Complex.of(Float);
C0:            ComplexFloat.new(0, 0);
```

The Complex class defined here is a more traditional generic class, which uses NumType as a type parameter. The real and imag members are both declared to be of type NumType, so they are guaranteed to be type-compatible.

The main difference between this class and its C++ or Java equivalent is that NumType, real, and imag are all declared in the same way. There is nothing in this declaration (such as a **template** or **typedef**) to indicate that NumType is a type parameter, while real and imag are data members. The difference between the three lies in the way they are used, not the way they are declared.

This declaration does establish an internal dependency between NumType real/imag. When NumType is specialized from Number to Float, the types of real and imag must be updated accordingly. Such updates of internal dependencies are a necessary consequence of using *virtual types*. [17] [9]

Virtual types are declared just like virtual methods, but their implementation is more difficult. Virtual methods use late binding, which means that the choice of which method to call is deferred until run-time. This is generally accomplished by storing methods in a virtual method table, and doing pointer lookups at run-time to select the correct one. In C++, the act of declaring a new class will create a new virtual method table.

Virtual types, on the other hand, have compile-time dependencies associated with them that cannot be deferred. When a virtual type is overridden, all variable and method signatures that rely upon that type will change. Like C++ template instantiations, binding a virtual type to a new definition will thus force a re-compilation of any affected code. The only alternative to recompilation is to abandon static type safety and have the compiler insert run-time type checks.

The Ohmu language actually uses both mechanisms. When a generic class is specialized at compile type, like ComplexFloat above, the compiler will recompile and statically type-check the new version. If the specialization is done at run-time, such “type binding” will be deferred by inserting run-time type checks. Since most new classes are declared at compile-time, run-time checks are seldom required.

4 Partial Evaluation

The issue of virtual types highlights another facet of using prototypes, and that is that there is no clear difference in Ohmu between run-time and compile-time code. In a traditional language, function calls and class instantiation are run-time operations, whereas inheritance and template instantiation are compile-time operations. Since Ohmu uses a single operation, binding time is undefined.

In an interpreted language this is not an issue, because everything happens at run-time. In order to compile code, however, certain computations must be shifted to compile-time. The Ohmu language uses a *partial evaluation* engine to accomplish this task. [10]

Partial evaluation is the cousin of *lazy evaluation*, which is implemented in many functional languages. One major advantage of the functional programming style is that because there is no run-time state, the time when computations happen is irrelevant. During lazy evaluation, computation is deferred until a later time. There are many cases, such as the infinite lists described earlier, where this can be quite useful; computationally expensive (or even infinite) calculations are only performed "as needed", and may be avoided altogether.

Partial evaluation could also be called "greedy evaluation" — the opposite of lazy evaluation. It works by locating invariant data, and performing computations with that data immediately at compile-time. In Ohmu, both constants and abstract prototypes are regarded as invariant. In fact, all the examples I have given up to this point have been compile-time operations. A declaration such as:

```
x: Integer;
```

defines x as an *abstract* integer; it does not define x as a variable. Any attempts to modify the value of x will generate an error. The parameter x can be bound to a subtype in derived structures, but parameter binding is not a destructive operation. The declaration

```
origin: Point.new(0,0);
```

creates a new object origin in which x and y have been redefined; it does not modify Point.x or Point.y. Note that the origin object declared here is a constant; origin.x and origin.y are permanently bound to 0, and that binding is performed at compile-time.

Run-time operations are those which involve variables. A variable in Ohmu is declared with a range and an initial value. The value may change over the course of execution, but it is constrained to be a subtype of the range:

```
x: Integer => 0;   // integer variable
...
x := 1;            // set x to 1
```

Any Ohmu expressions which involves variables will be deferred until run-time because such expressions are time-dependent. The result computed by a variable expression depends on exactly when it is evaluated in the course of executing the program.

Expressions which involve only constants and abstract types are known as *invariant expressions*. Since such expressions don't read from any variables, it does not matter when they are evaluated. The Ohmu compiler includes an full-fledged interpreter, and it will pre-compute all invariant expressions by invoking the interpreter at compile-time. For example:

```
x: 0;
y: 1;
z: Integer => 2;

a: sqrt(x*x + y*y);    // (compile-time) = 1
b: sqrt(x*x + z*z);    // (run-time) = sqrt(z*z)

myPoint:       Point.new(x,y);      // compile-time
herPoint:      Point.new(x,z);      // run-time
ComplexFloat:  Complex.of(Float);   // compile-time
```

Run-time expressions use lazy evaluation, and the result of evaluating such an expression will always take the current values of variables into account:

```
do {
  z := 1;
  print(b);            // prints 1  i.e. sqrt(1)
  print(herPoint.r);   // prints 1
  z := 2;
  print(b);            // prints 2  i.e. sqrt(4)
  print(herPoint.r);   // prints 2
};
```

Run-time expressions such as b and herPoint in the example above act like simple functions that take no arguments; they will be re-evaluated each time they are called. The semantics of run-time expressions and compile-time expressions are exactly the same; it's just that the value of a compile-time expression will never change because its arguments never change.

Note also that in order to set the value of z, we must place the statement z := 1 into an imperative statement list. Since expressions can be evaluated in any order, expressions are not allowed to change the state of a program. Only statements, which are ordered, are allowed to modify state. C-style calls such as sqrt(x++) are not allowed.

4.1 Structure Transformations

Partial evaluation becomes even more powerful when it is combined with structure transformations. Consider the following example:

```
foo: Struct {
  x,y: static Float => 0;
  r:   sqrt(x*x + y*y);
};

bar: transform foo {
  x,y: static Float => 1;
}

foo.r;   // r = 0, compile-time
bar.r;   // r = 1.41... compile-time
```

In this example, foo defines x and y as constants. The **static** keyword freezes a variable so that it can no longer vary. Ohmu provides several such keywords to allow fine-tuning of the partial evaluation process.

The reason for using “static variables” (yes, it’s an oxymoron) is that a variable (whether **static** or not) can be specialized in derived classes to any other value within the same range. If foo.x were declared as a simple 0 it could never be changed; no other number is a subtype of 0. Static variables are thus an implementation of “virtual constants.”

In any case, foo.r will be partially evaluated because it is an invariant expression. Yet when bar redefines x and y, that will affect the value of bar.r. The bar structure cannot simply inherit r as-is; the partial evaluation engine must detect the change and re-evaluate all relevant expressions.

This is the same problem I discussed in section 3.6 with regard to virtual types. Internal dependencies can take many forms. In general, whenever the partial evaluation engine reduces an expression, it creates an internal dependency, and the partial evaluator must record such dependencies so that it can update them if the relevant parameters are overridden in derived structures. Fortunately, this dependency tracking is only necessary at compile-time, and it incurs no run-time overhead.

4.2 Virtual Classes

In addition to supporting virtual types, Ohmu supports virtual classes. [13] A virtual class is simply a class definition that is nested inside another class:

```
List: Struct {
  DataType: Object;          // type parameter
  of:       bind(DataType);

  Node: Struct {
    item:   DataType;
    next:   Node;
  };

  begin: Node;
};
```

This is an alternate definition of a linked list class which behaves like a container instead of a stream. It encapsulates the list behavior by declaring an internal Node class. Since Node is a virtual class, it can be overridden just like any other parameter:

```
DoubleList: Struct {
  extends List {
    Node: Struct {           // specialize Node
```

```
        extends parent.Node;   // parent refers to List
        prev: Node;            // add another parameter
      };
    };

    end: Node;
  };
```

This definition of a doubly linked list simply inherits from the singly-linked version. It adds a prev parameter to Node, and an end parameter to the container. Note the use of the **parent** keyword, which refers the the original definition of Node. It is a bad idea to refer to a base class by name, because doing so hard-codes the structure of the inheritance hierarchy and thus prevents the use of mixins. [6] Using **parent**.Node instead of List.Node also resolves some subtle errors related to lexical scope that can crop up when specializing a virtual class.

Other than that, there is nothing special about this definition of a doubly linked list; it is simple and easy to understand. It is also impossible to write in a traditional OO language like Java or C++. There are two internal dependencies here: Node.prev and List.begin, which are both declared as type Node. Since C++ does not support virtual types, it cannot update type dependencies. If this code were written in C++, DoubleList would inherit the original type declarations for Node.next and begin. In other words, although Node.prev and DoubleList.end would refer to doubly-linked nodes, Node.next and DoubleList.begin would still refer to singly-linked nodes, and it would necessary to constantly downcast from List.Node to DoubleList.Node in order to traverse the list.

Despite the fact that lists are among the most basic of all data structures, standard OO inheritance cannot cope with a simple inheritance relationship between singly-linked lists and doubly-linked lists. This sort of headache points to a fundamental flaw in current OO languages – the fact that inheritance can only deal with virtual methods, not virtual types. By unifying functions and classes, this problem can be resolved.

5 Conclusion

The Ohmu language is interesting from a theoretical standpoint because it unifies the functional and object-oriented paradigms. Instead of tacking classes and inheritance onto a functional language as separate constructs, it extends the traditional typed lambda calculus so that OO concepts can be represented in natural way. A class is a second-order function. An instance is the activation record of a function. OO inheritance can be modeled, in part, as a variation of function currying.

It should be noted that the use of a partial evaluation engine also provides a natural division between the imperative and functional programing paradigms. Code that is partially evaluated at compile-time must use a pure functional style, because there is not yet any run-time state to modify. Run-time code is free to use imperative concepts.

Unifying functional and OO paradigms also has practical application, because it simplifies several design patterns. [7] For example, the Abstract Factory and Factory Method design patterns are necessary only because traditional OO classes cannot be virtual. In Java and C++, all classes must be specified at compile-time. The factory design patterns are forced to hide class instantiation behind virtual methods so that the choice of which class to instantiate can be deferred until run-time. The Ohmu language supports virtual classes and class parameters natively, so this extra level of indirection is unnecessary. The Prototype design pattern has likewise been incorporated directly into the type system.

In a more general sense, unifying methods and classes makes the Ohmu language scale-independent. In a traditional OO language, methods are combined into classes, classes are combined into frameworks, frameworks are combined into libraries, etc. At each level of the hierarchy, the programming constructs and operations change, and this makes it difficult to create complex, large-scale programs.

This is a general problem with mainstream OO languages. Object-oriented inheritance is a powerful tool for manipulating individual classes, but most solutions, including almost all design patterns, require a framework of interacting classes. Mainstream OO languages do not provide any real mechanism for manipulating such frameworks. [15] As a result, a whole slew of new programming paradigms, such as aspect-oriented programming, [12] component-based programming, [1] feature-based programming, etc. have been proposed to provide operations on class frameworks.

If classes can be treated as functions, however, then a set of interacting classes is no different from a set of interacting methods. A set of interacting methods, in turn, is just a class — the precise construct that standard object-oriented programming is designed to handle. Unifying functions and classes thus resolves a major outstanding problem that has crippled the OO programming paradigm.

Bibliography

[1] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology, 1(4):355-398, October 1992.

[2] G. Bracha and W. Cook. Mixin-Based Inheritance. Joint ACM Conference on OOPSLA and ECOOP, 1990.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past, Adding Genericity to the Java Programming Language. Proceedings of OOPSLA '98.

[4] Guiseppe Castagna. Covariance and Contravariance: Conflict Without a Cause. ACM Transactions on Programming Languages and Systems, 1995.

[5] Czarnecki and U. Eisenecker. Generative Programming: Methods, Techniques, and Applications. Addison-Wesley, 2000.

[6] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. ACM Symposium on Principles of Programming Languages, pages 171-183, 1998.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Mass. 1995.

[8] Paul Hudak, John Peterson, Joseph H. Fasel. A Gentle Introduction to Haskell, available at http://www.haskell.org/tutorial

[9] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. Technical report, University of Pennsylvania, 1998.

[10] Neil Jones, Carsten Gomard, and Peter Sestoft. Partial Evaluation and Automatic Program Generation. Prentice Hall, 1993.

[11] G. Kiczales, J. des Rivieres, and D. G. Bobrow. The Art of the Metaobject Protocol. The MIT Press, Cambridge, MA, 1991.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopez, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. Proceedings of ECOOP '97.

[13] O.L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. Proceedings of OOPSLA '89.

[14] Ole Lehrmann Madsen, Birger Møller-Pedersen, Kristen Nygaard. Object-Oriented Programming in the BETA Programming Language. Addison-Wesley and ACM Press, 1993 ISBN 0-201-62430-3

[15] Gail Murphy and David Notkin. The Interaction Between Static Typing and Frameworks. Technical Report TR-93-09-02, University of Washington, 1993. See also: The Use of Static Typing to Support Operations on Frameworks. Object-Oriented Systems 3, 1996, pp. 197-213.

[16] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. Proceedings of ECOOP, 1998.

[17] K. K. Thorup and M. Torgersen. Unifying Genericity — Combining the Benefits of Virtual Types and Parameterized Classes. Proceedings of ECOOP, 1999.

An Analysis of Constrained Polymorphism for Generic Programming

Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock

Open Systems Lab
Indiana University
Bloomington, IN USA
{jajarvi,lums,jsiek,jewillco}@osl.iu.edu

Abstract. Support for object-oriented programming has become an integral part of mainstream languages, and more recently generic programming has gained widespread acceptance. A natural question is how these two paradigms, and their underlying language mechanisms, should interact. One particular design option, that of using subtyping to constrain the type parameters of generic functions, has been chosen for the generics extensions to Java and C#. The leading alternative to subtype-based constraints is to use type classes, as they are called in Haskell, or concepts, as they are called in the C++ generic programming community. In this paper we argue that while object-oriented interfaces and concepts are similar in many ways, they also have subtle but important differences that make concepts more suitable for constraining polymorphism in generic programming.

1 Introduction

Generic programming is an emerging programming paradigm for writing highly reusable libraries of algorithms. The generic programming approach has been used extensively within the C++ community, in libraries such as the Standard Template Library [31, 30], Boost Graph Library [28] and Matrix Template Library [29]. Generic algorithms are parameterized with respect to the types of their arguments so that a single implementation may work on a broad class of different argument types.

For modularity, it is important for generic functions to be type checked separately from their call sites. The body of a generic function should be type checked with respect to its interface, and a call to that function should be type checked with respect to the same interface. Separate type checking helps the author of the generic function to catch errors in its interface and implementation, and more importantly, provides better error messages for incorrect uses of generic functions.

To provide separate type checking, a programming language must have a mechanism for constraining polymorphism. Several mainstream object-oriented languages with support, or proposed support, for generics, such as Java, C#, and Eiffel, implement variations of *F-bounded polymorphism* [6]. Haskell, a modern functional language, uses *type classes* [34] as the constraint mechanism for polymorphic functions. ML has parameterized modules, called functors, whose parameters are constrained by *signatures*. Other approaches include *where clauses* in CLU [23]. C++ is an example of a language without built-in support for constraints, and which has no direct support for separate type checking: the body of a generic function is type checked at each call site.

In our recent study [14], we evaluated six mainstream programming languages with respect to their support for generic programming. Mainstream object-oriented languages did not rank highly in this evaluation; practical problems encountered include verbose code, redundant code, and difficulties in composing separately defined generic components. These problems relate to the constraint mechanisms used in the various languages. Consequently, this paper focuses on the suitability of different constraint mechanisms for use in generic programming. We analyze current manifestations of subtype-bounded polymorphism in mainstream object-oriented languages, as well as other constraint mechanisms proposed in the literature, and identify the causes of the above problems. We argue that the current implementations of subtype-based constraint mechanisms in mainstream object-oriented languages are a major hindrance to effective generic programming; it proves difficult to organize constraints into well-encapsulated abstractions. We describe how object-oriented languages could be adapted to avoid the problems mentioned above. The inspiration for the proposed changes comes from constraint mechanisms such as those in Haskell and ML, which are not affected by these problems.

2 Background

We start with a short description of generic programming, and then describe two families of type systems/language mechanisms for supporting generic programming. The first family is based on just parametric polymorphism whereas the second family is based on subtype-bounded parametric polymorphism.

2.1 Generic programming

Generic programming is a systematic approach to software reuse. In particular, it focuses on finding the most general (or abstract) formulations of algorithms and then efficiently implementing them. These two aspects, generality and efficiency, are opposing forces, which is perhaps the most challenging aspect of this practice. The goal is for a single algorithm implementation to be usable in as many situations as reasonably possible without sacrificing performance. To cover all situations with the best performance, it is often necessary to provide a small family of generic algorithms with automatic dispatching to the appropriate implementation based on the properties of the input types.

There are several ways in which an algorithm can be made more general. The simplest and most common method is to parameterize the types of the elements the algorithm operates on. For example, instead of writing a matrix multiply function that works only for matrices of ***double***, one can parameterize the function for matrices of any numeric type. Another way in which algorithms can be parameterized is on the representations of the data structures they manipulate. For example, a linear search function can be generalized to work on linked lists, arrays, or indeed any sequential data structure provided the appropriate common interface can be formulated. Yet another approach to generalization is to parameterize certain actions taken by the algorithm. For example, in the context of graph algorithms, a breadth-first search (BFS) algorithm can invoke a user-defined callback function when tree edges are discovered. A client could use this to record the parent of each node in a BFS tree for the graph. The end result of this abstraction process should be an algorithm that places the minimum number of requirements on its input while still performing the task efficiently.

Terminology Fundamental to realizing generic algorithms is the notion of abstraction: generic algorithms are specified in terms of abstract properties of types, not in terms of particular types. In the terminology of generic programming, a *concept* is the formalization of an abstraction as a set of requirements on a type (or on several types) [18, 1]. These requirements may be semantic as well as syntactic. A concept may incorporate the requirements of another concept, in which case the first concept is said to *refine* the second. Types that meet the requirements of a concept are said to *model* the concept. Note that it is not necessarily the case that the requirements of a concept involve just one type; sometimes a concept involves multiple types and specifies their relationships.

A concept consists of four different types of requirements: associated types, function signatures, semantic constraints, and complexity guarantees. The *associated types* of a concept specify mappings from the modeling type to other collaborating types (such as the mapping from a container to the type of its elements). The function signatures specify the operations that must be implemented for the modeling type. A *syntactic concept* consists of just associated types and function signatures, whereas a *semantic concept* also includes semantic constraints and complexity guarantees [18]. At this point in the state of the art, type systems typically do not include semantic constraints and complexity guarantees. For this paper we are only concerned with syntactic concepts, so "concept" will mean "syntactic concept."

Generic programming requires some kind of polymorphism in the implementation language to allow a single algorithm to operate on many types. The remainder of this section reviews different language mechanisms and type systems that support polymorphism.

2.2 Parametric polymorphism

Generic programming has its roots in the higher-order programming style commonly used in functional languages [19]. The following ***find*** function is a simple example of this style: functions are made more general by adding function parameters and type parameters. In this example we parameterize on the ***T*** and ***Iter*** types and pass in functions for comparing elements (***eq***) and for manipulating the iterator (***next***, ***at_end***, and ***current***). This style obtains genericity using only unconstrained parametric polymorphism. For purposes of discussion we take the liberty of extending C# with polymorphic functions, function types, and type aliases as class members.

```
Iter find<Iter>(Iter iter, Iter.value_type x, ((Iter.value_type, Iter.value_type) → bool) eq,
                (Iter → Iter) next, (Iter → bool) at_end, (Iter → Iter.value_type) current)
{
  for (; !at_end(iter); iter = next(iter)) {
    Iter.value_type y = current(iter);
    if (eq(x, y))
      break;
  }
  return iter;
}

bool int_eq(int a, int b) { return a == b; }

class ArrayIterator<T> {
  typedef T value_type; ...
}

ArrayIterator<T> array_iter_next<T>(ArrayIterator<T> iter) { ... }

bool array_iter_at_end<T>(ArrayIterator<T> iter) { ... }

T array_iter_current<T>(ArrayIterator<T> iter) { ... }

void main() {
  int[] array = new int[]{1, 2, 3, 5};
  ArrayIterator<int> i(array);
  i = find(i, 2, int_eq, array_iter_next, array_iter_at_end, array_iter_current);
}
```

This example demonstrates one obvious disadvantage of the higher-order style: the large number of parameters for ***find*** makes it unwieldy to use. One solution to this problem is to introduce where clauses (various forms of which can be found in CLU [23], Theta [11], and Ada [33]). A where clause is a list of function signatures in the declaration of a generic function which are automatically looked up at each call site and implicitly passed into the function. This makes calling generic functions less verbose.

```
Iter find<Iter, T>(Iter iter, T x)
  where bool eq(T, T), Iter next(Iter), bool at_end(Iter), T current(Iter)
{ ... }

bool eq(int a, int b) { return a == b; }

class ArrayIterator<T> { ... }

ArrayIterator<T> next<T>(ArrayIterator<T> iter) { ... }

bool at_end<T>(ArrayIterator<T> iter) { ... }

T current<T>(ArrayIterator<T> iter) { ... }

void main() {
  int[] array = new int[]{1, 2, 3, 5};
  ArrayIterator<int> i(array);
  i = find(i, 2);
}
```

The addition of where clauses is not a fundamental change to the type system of the language; it is syntactic sugar for explicitly passing the function arguments.

2.3 Concepts

Similar sets of requirements often appear in many generic functions, so grouping related requirements together has software engineering benefits. For example, in a generic library such as the C++ Standard Library, all functions on sequences include requirements on their iterator parameters. Where clauses do not provide a way to group and reuse requirements. This is the role played by concepts. In the following example we create two concepts: one for expressing the comparison requirement, and one for grouping together the iterator operations. We are again using the base syntax of C#, but this time extended with concepts (we define the semantics of concepts later in this section).

```
concept Comparable<T> {
  bool eq(T, T);
}

concept Iterator<Iter> {
  type Iter.value_type; // Require an associated type

  Iter next(Iter);
  bool at_end(Iter);
  value_type current(Iter);
}

Iter find<Iter, T>(Iter iter, T x)
  where T models Comparable,
        Iter models Iterator,
        Iterator(Iter).value_type == T
{ ... }
```

A model of a concept is a set of types and a set of functions that meet the requirements of the concept. Some languages link implementations to concepts through an explicit *models declaration* (cf. Haskell instance declarations). At the call site for ***find***, for each concept requirement, a corresponding models declaration must be found.

```
int models Comparable {
  bool eq(int a, int b) { return a == b; }
}

class ArrayIterator<T> { ... }

forall<T> ArrayIterator<T> models Iterator {
  type value_type = T;
  ArrayIterator<T> next(ArrayIterator<T> iter) { ... }
  bool at_end(ArrayIterator<T> iter) { ... }
  value_type current(ArrayIterator<T> iter) { ... }
}

void main() {
  int[] array = new int[]{1, 2, 3, 5};
  ArrayIterator<int> i(array);
  i = find(i, 2);
}
```

The expression ***Iterator(Iter).value_type*** in the constraints for ***find*** accesses the ***value_type*** type definition from within the models declaration for ***Iter***. This mechanism provides a way to map from the primary types of the concept to the associated types.

Analogously to inheritance, concepts can be built from other concepts using refinement. A simple example of this is the following ***BidirectionalIterator*** concept.

```
concept BidirectionalIterator<Iter> : Iterator<Iter> {
  Iter prev(Iter);
}
```

One important observation about concepts is that they are not types. They can not be used as the type of a parameter, or to declare a variable. For the mathematically oriented, a concept is a set of multi-sorted algebras [18]. Roughly speaking, a multi-sorted algebra corresponds to a module: it is a collection of data types (the sorts) and functions (the operations of the algebra). Earlier we defined a concept as requirements on one or more types. The correspondence between these two definitions is the classic identification of a set with the predicate that specifies which elements are in the set (the elements in this case are modules).

In practice it is convenient to separate the data types of a module into two groups: the main types and the associated types. An example of this is an iterator (the main type) and its element type (an associated type). In a generic algorithm such as ***find***, a common need is the ability to obtain an associated type given the main type. A module then consists of a partial map from identifiers (names for associated types) to types $\mathrm{asc}(M) : \mathit{Id} \rightharpoonup \mathit{Type}$, and a partial map from function signatures (the name, parameter types, and result type) to function implementations $\Sigma(M) : S \rightharpoonup F$.

We formally define a concept C as a predicate on some *main types* $\vec{t}$ and a module M: $C(\vec{t}, M) = \mathcal{A} \wedge \mathcal{F} \wedge \mathcal{ST}$ where $\mathcal{A}$ is of the form $\vec{x} \subseteq \mathrm{dom}(\mathrm{asc}(M))$ (where $\vec{x}$ are the associated types required by C), $\mathcal{F}$ is of the form $\vec{s} \subseteq \mathrm{dom}(\Sigma(M))$ (where $\vec{s}$ are the function signatures required by C), and $\mathcal{ST}$ is of the form $\tau_1 = \tau'_1 \wedge \cdots \wedge \tau_n = \tau'_n$ (where the τ_i and τ'_i for $i = 1 \ldots n$ are pairs of type expressions which are required to be equal). The following is the ***Iterator*** concept expressed using this notation:

```
Iterator(Iter,M) ≡
  {value_type} ∈ dom(asc(M)) ∧
  { next : Iter → Iter, at_end : Iter → bool, current : Iter →
asc(M)(value_type) } ⊆ Σ(M)
```

In the previous example, the body of the models declaration

```
forall<T> ArrayIterator<T> models Iterator {
  type value_type = T;
  ArrayIterator<T> next(ArrayIterator<T> iter) { ... }
  bool at_end(ArrayIterator<T> iter) { ... }
  value_type current(ArrayIterator<T> iter) { ... }
}
```

can be viewed as a parameterized module with the following set of function signatures:

```
ArrayIterModule ≡ Λ T.
({(value_type, T)},
 {
   next : ArrayIterator<T> → ArrayIterator<T> = ...,
   at_end : ArrayIterator<T> → bool = ...,
   current : ArrayIterator<T> → value_type = ...
 })
```

So for any type ***T***, ***Iterator(ArrayIterator<T>, ArrayIterModule<T>)*** is true. We formally define that a sequence of types $\vec{t}$ together with a module M models a concept c when $c(\vec{t}, M)$ is true. We often say that a sequence of types models a concept, leaving out mention of the module of functions. This abbreviated form is written $c(\vec{t})$ and means that there is a models declaration in scope that associates a set of associated types and functions with the types $\vec{t}$ and concept c.

A concept c refines another concept c', denoted by $c \preceq c'$, if $\forall \vec{t}, m.\ c(\vec{t}, m)$ implies $c'(\vec{t}, m)$.

To describe concept-bounded types (and later subtype-bounded) we use the general setting of *qualified types* [16] to allow for a more uniform presentation. A qualified type is of the form $P => \tau$ where P is some predicate expression and τ is a type expression. The intuition is that if P is satisfied then $P => \tau$ has type τ. A qualified polymorphic type is then written

$$\forall t.\ P => \tau \tag{1}$$

or with multiple type parameters

$$\forall \vec{t}.\ P => \tau \tag{2}$$

A concept-bounded type is a qualified type where the predicates are models assertions. So concept-bounded polymorphic types have the following form.

$$\forall \vec{t}.\ c_1(\vec{t_1}) \wedge \cdots \wedge c_n(\vec{t_n}) => \tau \tag{3}$$

where $\vec{t_i} \subseteq \vec{t}$, the c_i's are concepts, and τ is a type expression possibly referring to types in $\vec{t}$.

The above definitions describe the structural aspects of modeling and refinement. However, languages such as Haskell and the extended C# of this paper use nominal conformance. That is, in addition to the structural properties being satisfied, there must also be explicit declarations in the program to establish the modeling and refinement relations.

Related constraint mechanisms Haskell and ML provide constraint mechanisms that share much in common with concepts. The following example, written in Haskell, groups the constraints from the previous example into type classes named ***Comparable*** and ***Iterator*** and then uses them to constrain the ***find*** (Haskell is a functional language, not object-oriented, and does not have object-oriented-style classes). In the declaration for ***find***, the ***Comparable T*** ⇒ part is called the "context" and serves the same purpose as the CLU where clause. The ***Int*** type is made an *instance* of ***Comparable*** by providing a definition of the required operations. In generic programming terminology, we would say that ***Int*** models the ***Comparable*** concept. Note that Haskell supports multi-parameter type classes, as seen in the ***Iterator*** type class below. The syntax ***i*** → ***t*** below means that the type ***t*** is functionally dependent on ***i***, which is how we express associated types in Haskell.

```
class Comparable t where
  eq :: t → t → Bool

class Iterator i t | i → t where
  next :: i → i
  at_end :: i → Bool
  current :: i → t

find :: (Comparable t, Iterator i t) ⇒ i → t → i
find iter x =
  if (at_end iter) || eq x (current iter) then
    iter
  else
    find (next iter) x

instance Comparable Int where
  eq i j = (i == j)
```

The ***instance*** declarations can be more complex. For example, the following *conditional* instance declaration makes all lists ***Comparable***, as long as their element types are ***Comparable***:

```
instance Comparable t ⇒ Comparable [t] where
  ...
```

ML *signatures* are a structural constraint mechanism. A signature describes the public interface of a module, or *structure* as it is called in ML. A signature declares which type names, values (functions), and nested structures must appear in a structure. A signature also defines a type for each value, and a signature for each nested structure. For example, the following signature describes the requirements of ***Comparable***:

```
signature Comparable =
sig
  type ElementT
  val eq : ElementT →ElementT →bool
end
```

Any structure that provides the type ***ElementT*** and an ***eq*** function with the appropriate types conforms to this signature without any explicit instance declarations. For example:

```
structure IntCompare =
struct
  type ElementT = int
  fun eq i1 i2 = ...
end
```

2.4 Subtype-bounded polymorphism

For object-oriented languages, the subtype relation is a natural choice for constraining generic functions. This section describes the various forms of subtype-bounded polymorphism that appear in mainstream languages and in the literature.

Bounded quantification Cardelli and Wegner [7] were the first to suggest using subtyping to express constraints, and incorporated *bounded quantification* into their language named Fun. The basic idea is to use subtyping assertions in the predicate of a qualified type. For bounded quantification the predicates are restricted to the form $t \leq \sigma$ where t is a type variable and σ does not refer to t. So we have polymorphic types of the form

$$\forall t.\ t \leq \sigma => \tau[t] \tag{4}$$

where t is a type variable, σ is a type expression that does not refer to t, and $\tau[t]$ is a type expression τ that may refer to t.

Fun is an unusual object-oriented language in that subtyping is structural, and there are no classes or objects; it has records, variants, and recursive types. The idea of bounded quantification carries over to mainstream object-oriented languages, the main change being the kinds of types and subtyping relations in the language. Subtyping in languages such as C++, Java, and C# is between classes (or between classes and interfaces). The following is an attempt to write the ***find*** example using bounded quantification. There are two options for how to write the ***eq*** method in the ***Int*** class below. The first option results in a type error because method parameters may not be covariant (Eiffel supports covariance, but its type system is unsound [9, 4]). The second option requires a downcast, opening the possibility for a run-time exception. This is an instance of the classic binary method problem [5].

```
interface Comparable {
  bool eq(Comparable);
}
Iterator find<T : Comparable>(Iterator iter, T x) { ... }
class Int : Comparable {
  bool eq(Int i) { ... } // Not a valid override
  bool eq(Comparable c) { ... } // Requires a downcast
}
```

F-bounded polymorphism Bounded quantification was generalized to *F-bounded polymorphism* by Canning et al. [6], which allows the left-hand side of a subtyping constraint to also appear in the right-hand side, thus enabling recursive constraints.

$$\forall t.\ t \leq \sigma[t] => \tau[t] \tag{5}$$

Types that are polymorphic in more than one type can be expressed by nesting.

$$(\forall t_1.\ t_1 \leq \sigma[t_1] => (\forall t_2.\ t_2 \leq \sigma[t_1, t_2] => (\forall t_3.\ t_3 \leq \sigma[t_1, t_2, t_3] => \tau[t_1, t_2, t_3])))$$

However, a constraint on type t_i may only refer to t_i and earlier type parameters.

The following example shows the ***find*** example, this time written using F-bounded polymorphism. We can now express the program without downcasts.

```
interface Comparable<T> {
  bool eq(T);
}
```

```
interface Iterator<Iter,T> {
  Iter next();
  bool at_end();
  T current();
}
Iter find<T, Iter>(Iter iter, T x)
  where T : Comparable<T>,
        Iter : Iterator<Iter,T>
{ ... }
class Int : Comparable<Int> {
  bool eq(Int i) { ... }
}
```

F-bounded polymorphism in turn was generalized to systems of mutually recursive subtyping constraints by Curtis [10, 12]. A *recursively subtype-constrained type* is of the form $P => \tau$ where P is a predicate of the form $\tau_1 \leq \tau_1' \wedge \cdots \wedge \tau_n \leq \tau_n'$. Then a recursively constrained polymorphic type is of the form

$$\forall \vec{t}.\ \tau_1 \leq \tau_1' \wedge \cdots \wedge \tau_n \leq \tau_n' => \tau \tag{6}$$

where the type variables in $\vec{t}$ can appear anywhere in the type expressions τ_i, τ_i', and τ. Recursively constrained polymorphic types, with some minor restrictions, are used in the generics extensions for Java and C#.

The following is an example of mutually recursive subtype constraints. The interface describing a graph node is parameterized on the edge type, and vice versa, and the ***breadth_first_search*** function uses the two interfaces in a mutually recursive fashion.

```
interface Node<E> {
  public List<E> out_edges();
}
interface Edge<N> {
  public N source();
  public N target();
}
public void breadth_first_search<N, E>(N n)
where N: Node<E>,
      E: Edge<N> { ... }
```

2.5 Definitions of the subtype relation

Subtype-bounded polymorphism expresses constraints based on the subtyping relation, so the expressiveness of the constraints is very much dependent on what types and subtype relations can be defined in the language. As mentioned in Section 2.4, much of the literature on bounded and F-bounded polymorphism [7, 6] used languages with records, variants, and recursive types and used a structural subtyping relation. Mainstream languages like C++, Java, and C# define subtyping as subclassing, a named subtyping relation between object types.

For a type ***B*** to be a subtype of some type ***A*** in a subtype relation that is based on structural conformance, ***B*** must have at least the same capabilities as ***A***. For example, if ***A*** is a record type, then ***B*** must have all the fields of ***A*** and the types of those fields must be subtypes of the corresponding fields in ***A***. A subtype relation based on named conformance, on the other hand, requires an explicit declaration in addition to the structural conformance requirement.

Mainstream object-oriented languages, such as C++, Java, C#, and Eiffel, unify subtyping with subclassing. The subtype relation is established at the point of definition of each class by declaring its superclasses. In particular, it is not possible to add a new supertype to an existing class without modifying the definition of the class. Mechanisms permitting such *retroactive subtyping* (or *retroactive abstraction*) declarations have been proposed and can be found in several programming languages, such as Sather [26, 27] and Cecil [8].

3 Discussion

This section discusses problems arising in object-oriented languages when attempting to follow the generic programming paradigm. Our earlier study in [14] showed that generic programming suffers from a set of distinct problems, whose cumulative effect is even more significant. As some of the symptoms, we observed verbose code in the form of excessive numbers of type parameters and constraints, awkward constructions to work around language limitations, difficulties in library maintenance, and the forced exposure of certain implementation details; the examples in [14] clearly demonstrate this.

We describe several extensions to Generic C# that lead to notably improved support for generic programming. We also describe a source-to-source translation of some of the extended features to current Generic C#.

3.1 Accessing and constraining associated types

Associated type constraints are a mechanism to encapsulate constraints on several functionally dependent types into one entity. Section 2.3 gave an example of an iterator concept and its associated type ***value_type***. As another example, consider the following two concepts specifying the requirements of a graph type. The ***IncidenceGraph*** concept requires the existence of vertex and edge associated types, and places a constraint on the edge type:

```
concept GraphEdge<Edge> {
  type Vertex;
  Vertex source(Edge);
  Vertex target(Edge);
}
concept IncidenceGraph<Graph> {
  type Vertex;
  type Edge models GraphEdge;
  Vertex == GraphEdge<Edge>.Vertex;
  type OutEdgeIterator models Iterator;
  Iterator<OutEdgeIterator>.value_type == Edge;

  OutEdgeIterator out_edges(Graph g, Vertex v);
  int out_degree(Graph g, Vertex v);
}
```

All but the most trivial concepts have associated type requirements, and thus a language for generic programming must support their expression. Of mainstream languages, ML supports this via types in structures and signatures; C++ can represent associated types as member typedefs or *traits classes* [25] but cannot express constraints on them. Java and C# do not provide a way to access and place constraints on type members of generic type parameters. However, associated types can be emulated using other language mechanisms.

```
interface GraphEdge {
  type Vertex;
  Vertex source();
  Vertex target();
}

interface IncidenceGraph {
  type Vertex;
  type Edge : GraphEdge;
  Vertex == Edge.Vertex;

  type OutEdgeIterator
    : IEnumerable<Edge>;

  OutEdgeIterator out_edges(Vertex v);
  int out_degree(Vertex v);
}
```

(a)

```
interface GraphEdge<Vertex1> {
  Vertex1 source();
  Vertex1 target();
}

interface IncidenceGraph<
  Vertex1, Edge1, OutEdgeIterator1>
  where
    Edge1 : GraphEdge<Vertex1>,
    OutEdgeIterator1 :
      IEnumerable<Edge1> {
  OutEdgeIterator1 out_edges(Vertex1 v);
  int out_degree(Vertex1 v);
}
```

(b)

Fig. 1. Graph concepts represented as interfaces which can contain associated types (a), and their translations to traditional interfaces (b).

A common idiom used to work around the lack of support for associated types is to add a new type parameter for each associated type. This approach is frequently used in practice. The C# ***IEnumerable<T>*** interface for iterating through containers serves as an example. When a type extends ***IEnumerable<T>*** it must bind a concrete value, the value type of the container, to the type parameter ***T***. The class ***AdjacencyList***, which extends the ***IncidenceGraph*** interface, in Figure 2(b) is an example of the same situation. The following generic function, which has ***IncidenceGraph*** as a constraint, includes an extra type parameter for each associated type. These type parameters are used as arguments to ***IncidenceGraph*** in the constraint on the actual graph type parameter.

```
G_Vertex first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>(G g, G_Vertex v)
  where G : IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator> {
  return g.out_edges(v).Current.target();
}
```

The main problem with this technique is that it fails to encapsulate associated types and constraints on them into a single concept abstraction. Every reference to a concept, whether it is being refined or used as a constraint by a generic function, needs to list all of its associated types, and possibly all constraints on those types. In a concept with several associated types, this becomes burdensome. In the experiment described in [14], the number of type parameters in generic algorithms was often more than doubled due to this effect.

A direct representation for associated types could be added to Generic C# as an extension, providing *member types* similar to those in C++. In this extension, interfaces can declare members which are placeholders for types, and place subtype constraints on these types. Classes extending these interfaces must bind concrete values to these types. As an example, Figure 1(a) shows two concepts from the domain of graphs. The ***GraphEdge*** concept declares the member type ***Vertex***. The ***IncidenceGraph*** concept has two associated types: ***Vertex*** and ***Edge***. Note the three constraints: ***Edge*** must be a subtype of ***GraphEdge***; ***Vertex*** must be the same type as the associated type, also named ***Vertex***, of ***Edge***; and ***OutEdgeIterator*** must conform to ***IEnumerable<Edge>***. The last constraint uses the standard ***IEnumerable*** interface which does not use the member type extension; the two styles can coexist.

This representation for associated types can straightforwardly be translated into the emulation using extra type parameters which was described earlier. Figure 1(b) shows the translated versions of the graph interfaces. In this translation, each interface containing associated types has an extra parameter added for each associated type. The subtype constraints on the associated types are converted to subtype constraints on the corresponding type parameters. In classes inheriting from such interfaces, the associated type definitions are converted to type arguments of the interfaces, as shown in Figure 2(b). Generic functions using interfaces with associated types also have an extra type parameter added for each associated type (Figure 3(b)). Within the body and constraints of a generic function, references to associated types are converted to references to the corresponding type parameters. Equality constraints between two types are handled by unifying, in the logic programming sense, the translations of the types required to be equal. For example, the type ***Vertex1*** is used both as the ***Vertex*** associated type for ***GraphEdge*** and for ***IncidenceGraph*** in Figure 1(b). Figure 2 shows the code defining two concrete classes which extend the interfaces for ***GraphEdge*** and ***IncidenceGraph***, both before and after translation. We used this translation of associated types, manually, while implementing the graph library described in [14].

The advantages of the associated type extension become evident when using interfaces to constrain type parameters of a generic algorithm. Consider the ***first_neighbor*** function in Figure 3. The function has two parameters: a graph and a vertex. Using the extension, shown in Figure 3(a), a single type parameter can describe the types and

```
class AdjListEdge : GraphEdge {
  type Vertex = int;
  ...
}

class AdjacencyList : IncidenceGraph {
  type Vertex = int;
  type Edge = AdjListEdge;

  type OutEdgeIterator =
    IEnumerable<AdjListEdge>;

  OutEdgeIterator out_edges(Vertex v) {...}

  int out_degree(Vertex v) {...}
}
```

(a)

```
class AdjListEdge : GraphEdge<int> {
  ...
}

class AdjacencyList
  : IncidenceGraph<int, AdjListEdge,
      IEnumerable<AdjListEdge> > {

  IEnumerable<AdjListEdge>
    out_edges(Vertex v) {...}

  int out_degree(Vertex v) {...}
}
```

(b)

Fig. 2. A concrete graph type which models the ***IncidenceGraph*** concept.

constraints of both of these parameters. In the translated code (Figure 3(b)), a separate type parameter is needed for each of the three associated types of the graph type.

Note that the translated code is not valid Generic C#; we are assuming that constraints on type parameters are propagated automatically from the interfaces which are used, which is not the case in the current version of Generic C#. Section 3.2 discusses this issue in more detail.

```
G.Vertex first_neighbor<G>(G g, G.Vertex v) where G : IncidenceGraph {
  return g.out_edges(v).Current.target();
}
```

(a)

```
G_Vertex first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>(G g, G_Vertex v)
  where G : IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator> {
  return g.out_edges(v).Current.target();
}
```

(b)

Fig. 3. A generic algorithm using ***IncidenceGraph*** as a constraint, both with (a) and without (b) the extension.

Note that an interface that contains associated types is not a traditional object-oriented interface; in particular, such an interface is not a type. As the translation suggests, these interfaces cannot be used without providing, either implicitly or explicitly, the values of their associated types. As a consequence, interfaces with associated types can be used as constraints on type parameters, but cannot be used as a type for variables or function parameters — uses that traditional interfaces allow. For example, the function prototype in Figure 3(a) cannot be written as:

```
IncidenceGraph.Vertex first_neighbor(IncidenceGraph g, IncidenceGraph.Vertex v);
```

The references to ***IncidenceGraph.Vertex*** are undefined; the abstract ***IncidenceGraph*** interface does not define a value for the ***Vertex*** associated type. This is a major difference between our translation and systems based on *virtual types* [24, 32]. In our translation, all associated types are looked up statically, and so the type of ***g*** is the interface ***IncidenceGraph***, not a concrete class which implements ***IncidenceGraph***. On the other hand, in systems with virtual types, member types are associated with the run-time type of a value, rather than its compile-time type; thus, the function definition above would be allowed. The virtual type systems described in [24, 32] do not provide means to express the constraints in the previous examples in type-safe manner. Ernst describes *family polymorphism* [13], a type-safe variation of virtual types, for the programming language Beta. This is a related mechanism to the extension proposed here for representing associated types in an object-oriented language. Whether family polymorphism can provide full support for associated types remains to be evaluated.

For the translation described here to work, it is important to be able to infer the values of associated types from the types bound to the main type parameters. This is not currently supported in Generic C# or Java generics. As an example of this, consider the following equivalent formulation of the ***first_neighbor*** function, which makes the use of the associated edge type more explicit:

```
G.Vertex first_neighbor<G>(G g, G.Vertex v) where G : IncidenceGraph {
  G.Edge first_edge = g.out_edges(v).Current;
  return first_edge.target();
}
```

In a call to ***first_neighbor***, a concrete graph type is bound to ***G***, and thus associated types, such as ***G.Edge***, can be resolved. In the translated version, however, it is less obvious that associated types can be inferred automatically:

```
G_Vertex first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>(G g, G_Vertex v)
  where G : IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator>
{
  G_Edge first_edge = g.out_edges(v).Current;
  return first_edge.target();
}
```

The two type parameters ***G_Edge*** and ***G_OutEdgeIterator*** are not the types of any of the function arguments, and thus are not directly deducible. To infer their types, the particular graph type used as ***G*** must be examined to find its associated type definitions. The associated types are expressed as type arguments to ***IncidenceGraph*** in an inheritance declaration. Inferring the associated types from constraints is possible in most cases, including all cases generated by the translation given here, but is not supported in the current proposals for Generic C# or Java generics.

3.2 Constraint propagation

In many mainstream object-oriented languages, the constraints on the type parameters to generic types do not automatically propagate to uses of those types. For example, although a container concept may require that its iterator type model a specified iterator concept, any generic algorithm using that container concept will still need to repeat the iterator constraint. This is done for error checking: instances of an interface must always be given correct type parameters, even within the definition of a generic method. The burden of this is that the check is done when a generic method is defined, rather than when it is used, and so the generic method ends up needing to repeat the constraints of all of the interfaces which it uses.

For example, without constraint propagation, the ***first_neighbor*** function from Figure 3(a) would need to be written as:

```
G.Vertex first_neighbor<G>(G g, G.Vertex v)
  where G : IncidenceGraph,
        G.Edge : GraphEdge,
        G.Edge.Vertex == G.Vertex,
        G.OutEdgeIterator : IEnumerable<G.Edge> {
  return g.out_edges(v).Current;
}
```

The problem with constraint propagation also applies to the translated version of ***first_neighbor*** (cf. Figure 3(b)):

```
G_Vertex first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>(G g, G_Vertex v)
  where G : IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator>,
        G_Edge : GraphEdge<G_Vertex>,
        G_OutEdgeIterator : IEnumerable<G_Edge> {
  return g.out_edges(v).Current;
}
```

The additional constraints in these examples merely repeat properties of the associated types of ***G*** which are already specified by the ***IncidenceGraph*** concept. This greatly increases the verbosity of generic code and adds extra dependencies on the exact contents of the ***IncidenceGraph*** interface, thus breaking the encapsulation of the concept abstraction.

This is not an inherent problem in subtype-based constraint mechanisms. For example, the Cecil language automatically propagates constraints to uses of generic types [8, § 4.2]. Constraint propagation is simple to implement: a naïve approach is to automatically copy the type parameter constraints from each interface to each of the uses of the interface.

3.3 Subclassing vs. subtyping

The subclass relation in object-oriented languages is commonly established in the class declaration, which prevents later additions to the set of superclasses of a given class. This is fairly rigid, and as many object-oriented languages unify subclassing and subtyping, the subtype relation is inflexible too. Several authors have described how this inflexibility leads to problems in combining separately defined libraries or components, and proposed solutions. Hölzle describes problems with component integration and suggests that adding new supertypes and new methods to classes retroactively, as well as method renaming, be allowed [15]. The Half & Half system [2] allows subtyping declarations that are external to class definitions, as do the Cecil [8] and Sather [26, 27] programming languages. Aspect oriented programming systems [21], such as AspectJ [20], can provide similar functionality by allowing modification of types outside of their original definitions. Structural subtyping does not suffer from the same problems. Baumgartner and Russo [3], as well as Läufer et al. [22], suggest adding a structural subtyping mechanism to augment the nominal subtyping tied to the inheritance relation.

Constraint mechanisms more directly supporting concepts, such as Haskell type classes and ML signatures, do not exhibit the retroactive modeling problem: instance declarations in Haskell are external to types, and ML signature conformance is purely structural.

The work cited above is in the context of object-oriented programming, but the use of the subtyping relation to constrain the type parameters of generic algorithms shares the same problems. If an existing type structurally conforms to the requirements of a generic algorithm, but is not a nominal subtype of the required interface, it can not be used as the type parameter of the algorithm. Current mainstream object-oriented programming languages do not provide a mechanism for establishing this relation; types cannot retroactively be declared to be models of a given concept. This problem of retroactive modeling is described further in [14]. The research cited above has demonstrated that retroactive subtyping can be implemented for an object-oriented language.

3.4 Constraining multiple types

Some abstractions define interactions between multiple independent types, in contrast to an abstraction with a main type and several associated types. An example of this is the mathematical concept ***VectorSpace*** (more examples can be found in [17]).

```
concept VectorSpace<V, S> {
  V models Field;
  S models AdditiveGroup;
  V mult(V, S);
  V mult(S, V);
}
```

For this example, it is tempting to think that the scalar type should be an associated type of the vector type. For example, the class ***matrix***<***float***> would only have ***float*** for its scalar type. However it also makes sense to form a vector space with ***matrix***<***float***> and ***vector***<***float***> as the vector and scalar types. So in general the scalar type of a vector space is not *determined* by the vector type.

It is cumbersome to express multi-parameter concepts using object-oriented interfaces and subtype-based constraints. One must split the concept into multiple interfaces.

```
interface VectorSpace_Vector<V, S> : AdditiveGroup<V> {
  V mult(S);
}
```

```
interface VectorSpace_Scalar<V, S> : Field<S> {
  V mult(V);
}
```

Algorithms that require the ***VectorSpace*** concept must specify two constraints now instead of one. For example:

```
Vector linear_combination_2<Vector, Scalar>(Scalar alpha1, Vector v1,
                                            Scalar alpha2, Vector v2)
  where Vector: VectorSpace_Vector<Vector, Scalar>,
        Scalar: VectorSpace_Scalar<Vector, Scalar>
{
  return alpha1.mult(v1).add(alpha2.mult(v2));
}
```

In general, if a concept hierarchy has height n, and places constraints on two types per concept, then the number of subtype constraints needed in an algorithm is 2^n, an exponential increase in the size of the requirement specification. Concept hierarchies of height from two to five are common in practice, and we have encountered even deeper hierarchies, but 2^5 is already a large number.

The constraint propagation extension discussed in Section 3.2 ameliorates this problem. The ***VectorSpace_Scalar*** interface is attached to the ***VectorSpace_Vector*** interface by the constraint on the type parameter ***S***:

```
interface VectorSpace_Vector<V, S> : AdditiveGroup<V>
  where S : VectorSpace_Scalar<V, S>
{
  V mult(S);
}
```

This prevents the exponential increase in the number of requirements, but the interface designer must still split up concepts in an arbitrary fashion. This problem could be overcome by an automatic translation of multi-parameter concepts into several interfaces, as done above. The ***linear_combination_2*** algorithm shown above needs only a single constraint now.

```
Vector linear_combination_2<Vector, Scalar>(Scalar alpha1, Vector v1,
                                            Scalar alpha2, Vector v2)
  where Vector: VectorSpace_Vector<Vector, Scalar> {
  return alpha1.mult(v1).add(alpha2.mult(v2));
}
```

4 Conclusion

The main contribution of this paper is to provide a detailed analysis of subtype-based constraints in relation to generic programming. We survey a range of alternatives for constrained parametric polymorphism, including subtype-based constraints in object-oriented languages. We identify problems that hinder effective generic programming in mainstream object-oriented languages, and pinpoint the causes of the problems. Some of the surveyed alternatives, such as concepts, ML signatures, and Haskell type classes, do not exhibit these problems. Based on these alternatives, we describe solutions that fit within the context of a standard object-oriented language. We describe an extension to C# that adds support for accessing and constraining associated types, constraint propagation, and multi-parameter concepts. We outline a translation of the extended features to the current Generic C# language.

Acknowledgments

We are grateful to Ronald Garcia for his comments on this paper. This work was supported by NSF grants EIA-0131354 and ACI-0219884, and by a grant from the Lilly Endowment. The fourth author was supported by a Department of Energy High Performance Computer Science Fellowship.

Bibliography

[1] M. H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[2] G. Baumgartner, M. Jansche, and K. Läufer. Half & Half: Multiple Dispatch and Retroactive Abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Ohio State University, 2002.

[3] G. Baumgartner and V. F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice and Experience*, 25(8):863–889, August 1995.

[4] K. B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Technical report, Williams College, 1996.

[5] K. B. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. F. Smith, V. Trifonov, G. T. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[6] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, 1989.

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

[8] G. Chambers and the Cecil Group. *The Cecil Language: Specification and Rationale, version 3.1*. University of Washington, Computer Science and Engineering, Dec. 2002. www.cs.washington.edu/research/projects/cecil/.

[9] W. R. Cook. A proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):304–311, 1989.

[10] P. Curtis. *Constrained quantification in polymorphic type analysis*. PhD thesis, Cornell University, Feb. 1990. www.parc.xerox.com/company/history/publications/bw-ps-gz/csl90-1.ps.gz.

[11] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA*, pages 156–158, 1995.

[12] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1. Elsevier, 1995.

[13] E. Ernst. Family polymorphism. In *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, June 2001.

[14] R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *OOPSLA*, Oct. 2003. To appear.

[15] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP*, volume 707 of *Lecture Notes in Computer Science*, pages 36–55. Springer, July 1993.

[16] M. P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[17] S. P. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[18] D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI–92–20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180, July 1992.

[19] A. Kershenbaum, D. Musser, and A. Stepanov. Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute, 1988.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, June 1997.

[22] K. Läufer, G. Baumgartner, and V. F. Russo. Safe structural conformance for Java. *Computer Journal*, 43(6):469–481, 2001.

[23] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[24] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA*, pages 397–406. ACM Press, 1989.

[25] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[26] S. M. Omohundro. The Sather programming language. *Dr. Dobb's Journal*, 18(11):42–48, October 1993.

[27] Sather home pages. www.icsi.berkeley.edu/~sather/.

[28] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library User Guide and Reference Manual*. Addison Wesley Professional, 2001.

[29] J. G. Siek and A. Lumsdaine. A modern framework for portable high performance numerical linear algebra. In *Modern Software Tools for Scientific Computing*. Birkhäuser, 1999.

[30] A. Stepanov. The Standard Template Library — how do you build an algorithm that is both generic and efficient? *Byte Magazine*, 20(10), Oct. 1995.

[31] A. A. Stepanov and M. Lee. The standard template library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, Apr. 1994. (http://www.hpl.hp.com/techreports).

[32] K. K. Thorup. Genericity in Java with virtual types. In *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471, 1997.

[33] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, February 1983.

[34] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.

Top-Down Decomposition in C++

Asher Sterkin and Avraham Poupko

NDS Technologies Israel Ltd.
P.O. Box 23012, Har Hotzvim
Jeruslaem 91235 Israel
asterkin@ndsisrael.com apoupko@ndsisrael.com

Abstract. This article describes how the Top-Down Decomposition approach, introduced by E.W. Dijkstra, can be implemented effectively, using the C++ programming language. A brief outline of the Top-Down Decompositions process, its advantages, and the most common misconceptions about this approach are presented. The role of C++ language features such as templates, and generic algorithms is discussed. Prime Number Generator exercise is used for illustrative purposes. The first version is implemented in the most straightforward way suitable for generating of 500 primes. The last version is fully optimized, making it possible to generate all primes within the 32-bit range.

1 Introduction

The objective of this paper is to demonstrate how the basic Top-Down Decomposition approach could be applied, using the C++ programming language, the C++ Standard Library, and Generic Programming approach.

The materials of this paper were used as a basis for an *Advanced C++ Programming* course taught by these authors, in the Hadassah Academic College, Jerusalem, as well as for an internal workshop on the same topics, held in NDS Technologies Israel Ltd.

2 Top-Down Decomposition

Top-Down Decomposition is a way of developing software through a number of iterative steps, in order to increase developer productivity and improve software quality. This approach was initially introduced by E.W. Dijkstra [1, 2] as a response to the so-called ”software crisis” of the late 1960s.

The Top-Down Decomposition process could be briefly outlined as follows:

1. Start from a dummy, ’do-nothing,’ program
2. Address the project organization and version control issues
3. At each step:
   (a) Implement the smallest possible piece of the program, either adding detail to the current level of abstraction or introducing a new one
   (b) Compile and test
   (c) If necessary, re-factor the code for optimization, bug fixes, and improved readability

   (d) Compile and test
4. Repeat the process until the whole program is completed

This approach provides the following advantages:

- High speed of development
- Small number of errors
- Early integration and testing
- Early demonstration/deployment
- Support for parallel development
- ’Just enough improvements’
- Strong feeling of ’flow’ and accomplishment

The main challenge in applying this approach is the identification of the levels of abstraction.

3 Top-Down Decomposition Misconceptions

The advantages of Top-Down Decomposition, detailed above, are regarded with skepticism by many software developers because they are wary of incurring the extra overhead required by introducing abstractions.

The Top-Down Decomposition approach encourages introducing abstraction layers such that, at each step either a new abstraction layer is introduced or a small portion of an already existing layer is detailed. Each new abstraction layer requires an interface between itself and the layer above it. The interface is represented as a set of data structures and functions for manipulating these data structures.

Software developers typically resist introduction of new layers (and their attendant functions), claiming that the layers incur too much overhead and citing concerns with run-time efficiency. Additionally, even when they do agree to introduce additional abstraction layers, they don’t adhere to the required separation between them - again citing efficiency. This laxity is exacerbated by allowing shortcuts and jumps through multiple layers - all the way to the lowest possible one.

These divergences are so ingrained in programming culture that software development folklore is full of stories about how the *purist* layering approach failed due to severe performance problems and how significantly better results were achieved with the *close to hardware* approach.

The problem with these claims is that they undermine the major goal of the Top-Down Decomposition: namely, controlled complexity and consequentially, flawless software. Perhaps the most damage caused by improper addressing of efficiency issues is to the very skill of using abstraction to fight complexity.

Although layers are used in modern software development practice in the form of the *architectural pattern* by that name[3], they address the enterprise class software coarse-grain structuring issues primarily in terms of portability, reuse and packaging. The whole number of layers is fixed and tends to be quite small (3-5). In other words, the Layers architectural pattern does help with the software organization but does not deal at all with the problem and solution themselves.

The point is that at a certain level of complexity the ability to properly introduce abstraction layers makes the whole difference between possible and impossible, rather than between nice and ugly or efficient and inefficient. The number of abstraction layers depends solely on the problem at hand and cannot be prescribed in advance.

Demonstrating how abstraction layers are introduced during the Top-Down Decomposition process will constitute the central theme of this paper.

As we explain below, the idea engrained in the minds of software developers-that the Top-Down Decomposition approach is ineffective, is not in fact an inherent flaw in the Approach. One of the main reasons that TDD is considered to be ineffective is the lack of proper language constructs.

The remainder of this paper attempts to show that proper use of C++ features, can allow a developer to achieve the advantages of TDD, without paying the price that is usually associated with layering. This will be done by presenting examples of correct Top-Down Decomposition, where indeed the reader can see that there is little or no runtime overhead.

4 The C++ Programming Language (Commonality and Variability)

The C++ programming language is a suitable choice due to its strong support for templates and inline functions, which almost completely eliminate the layering overhead. Appropriate usage of #define, templates and inline functions reduces the effective run time cost of many of the layers to near-zero. Additionally, their use makes it easier and less costly to enforce pure layer separation.

The basic assumption is that certain aspects of each program will inevitably change during its lifespan. It’s not a question of whether parts of the program will change, but ”which parts will change, and when: at run-time or at compile time?” On one hand, we want to make our programs as general as possible, meaning that our program can be configured at run time to support all kinds of behavior. This will eliminate the need to have different programs for different sets of data. On the other hand, the more assumptions we are able to make regarding the input, the less we need to evaluate at runtime, and the more efficient our program can run [4].

Making a decision where a particular variability point has to be reflected is a classical flexibility vs. efficiency trade-off. The greater the number of variability points that exist at the run-time, the more the computer resources (such as CPU and memory) will be consumed. Using the run-time only variability usually leads to over-designed systems. The irony is that when some resources are scarce all kinds of ugly optimization tricks are applied. This in turn would lead to hard coding of some variability points. Very often, these points are the very points that we would like to leave as run-time variants.

The C++ feature set encourages the developer to think in two dimensions simultaneously: compile time dimension and run-time dimension [4, 5]. For the later, the whole set of virtual functions, polymorphism and run-time (late) binding mechanisms is provided in order to achieve a required level of flexibility. C++ is not unique in this category. Other languages such as Java and C# supply the same level of service.

However, if changes were to occur only during compilation time, the great promise of the object-oriented approach may incur an unacceptable overhead for many real applications. That's where C++ templates, function overloading, static (early) binding, and inline functions become very handy. Today, these comprise unique features of C++ [1].

The good news about C++ is that it comes with a powerful standard library [6]. This library significantly simplifies and, to a certain degree, directs the Top-Down Decomposition process.

The C++ Standard Library contains a group of efficient template-based implementations of general-purpose containers such as *vector* and *list*, and a group of generic algorithms such as *copy*, *search*, and *sort*. Apparently, the C++ generic algorithms turn out to be the central part of the software development process, providing a set of generic skeletons that the application-specific code can be stuffed into. This meshes well with the Top-Down Decomposition process, as we shall see in the subsequent sections.

5 Prime Number Generator

In order to be able to demonstrate the Top-Down Decomposition approach we will use this evergreen exercise. Interesting enough, it has been always used for the demonstrating the Top-Down Decomposition approach by Dijkstra from his first paper on the subject [1].

6 Problem Statement

Compute and print a table of the first 500 prime numbers, arranged in 10 columns, each listing 50 prime numbers.

Definition 1. *An integer number is a prime if it can be divided wholly (with reminder 0) only by 1 and itself. Example: 3571. 0 and 1 are not considered to be primes.*

7 Step 1: Dummy, 'Do Nothing' Program

The first step is to write a dummy program, which does nothing or almost nothing and just helps with establishing the development environment. It's a good time for addressing the configuration management issues such as build system (make) and version control.

In the case of the Prime Number Generator its first dummy version will look as follows:

Although this piece of code indeed does almost nothing, it does provide a firm foundation for everything that follows. It embodies key decisions, such as coding standard, text editor, compiler, build-system and version control. Adopting the 'do-nothing'

[1] Using C++ does not come for free. Probably the most unfortunate fact about C++ is that's its syntax is littered with many complications, the major bulk of which are caused by the need to be backward compatible with "C". A simple, powerful programming language with a clean syntax is yet to come.

dummy program approach is therefore very important in helping make these decisions as early as possible in the development cycle. Postponing these decisions may entail much additional work and destabilize the software product.

8 Step 2: Implement the Main Loop

All but the most trivial programs include some form of iteration. In a case of GUI or Network programs it would be the so-called main message loop. In the case of the Prime Number Generator we're illustrating here, it would be a loop over the count of primes to be generated. At each iteration, one prime number is generated and printed. The most straightforward way to implement such a loop, using the C++ Standard Library, is as follows:

Note the use of `generate_n` algorithm for the prime number generation loop.

In [2] E.W. Dijkstra nominates three basic mental aids required for the software construction process: enumeration, mathematical induction and abstraction. In the context of to the Prime Number Generator we will deal with these topics in the following order: mathematical induction, abstraction, and enumeration (see Optimization).

8.1 Mathematical Induction

Mathematical induction has traditionally been associated either with repetitive programming constructs such as C++ *while*, *for* and *do-while* statements. In addition to their other benefits (see above) the C++ Standard Library algorithms can also help programmers to abstract out the mathematical induction. In the Prime Number Generator, the `generate_n` algorithm provides an abstraction of invoking a generic function Gen() N-times, sending the result to a generic iterator OutIt. Using this approach we introduce two new abstractions to be implemented at the next layer:

- Generator: to generate one prime at a time
- TabOutIterator: to print one prime number at a time, while applying all required table-formatting rules

8.2 Abstraction

The new abstraction layer is described in the Primes.h file:

In [2] E.W. Dijkstra strongly argues that each abstraction layer should be treated as a virtual machine, exposing a set of commands suitable for solving the problem of the higher abstraction layer. Three decades later Robert C. Martin has reinforced this approach by stating the *Dependency Inversion Principle* [7]. In simple words, *it's the responsibility of the lower layer to implement abstractions defined at the higher level*. Martin's Dependency Inversion Principle then in effect rejects the conception that the higher layer is condemned to resolving its problem in terms of abstractions dictated by the lower layer.

In the case of our Prime Number Generator, the *Primes* layer has to supply basic abstractions required for the main program layer for its task, namely generating and printing the first 500 primes

Each class of the Primes layer is declared in its own include file in order to facilitate version control and to reduce the impact of future changes. The Generator.h file for this step looks as follows:

In order to represent it as a layer the C++ namespace is used. C++ namespaces provide an effective means of encapsulation, without incurring any performance overhead. The Primes::Generator class functionality is implemented via a *self-operator*, effectively turning this class into a *functor* [6]. The Primes:: Generator class is parameterized with the type of prime number, which makes it generic for any underlying types of numbers. We shall see later how this parameterization helps with debugging and testing.

The TabOutIterator.h file for this step looks as follows:

The Primes::TabOutIterator class is also parameterized with the prime number type and encapsulates the table formatting functionality in the form of an *iterator* [6].

Despite its perceived generic nature this class in fact does belong to the Primes layer, which is responsible for supplying prime number formatting services to the upper layer. Even if such functionality were available from a 3rd party there are always some small specifics, which justify this pure layered approach (see the final implementation of TabOutIterator).

At the completion of this stage, we are able to compile and run the program. It will not format the table properly, nor will it generate prime numbers correctly (that will be handled in the following stages). But we nonetheless have made valuable progress.

9 Step 3: Formatting the Table

We now concentrate on properly formatting the table based on the number of columns. To achieve this the assignment operator has to be modified accordingly. This in turn would require introducing new TabOutIterator private member variables that well keep tract of column width, number of columns per line and index of the last number being printed. In order to initialize these variables properly the TabOutIterator constructor has to be modified. Finally, we have to handle a case when the last line of the table is incomplete. That could be done in the TabOutIterator's destructor.

The modified Primes::TabOutIterator class looks like as follows:

At this point we have a program that uses a generator to generate primes, and uses an output iterator to output them. We can now compile the program, and the output will be formatted correctly. We now need to implement the generator.

10 Step 4: Implementing the Generator

The Primes::Generator class does the following: for each execution of the *self-operator*, it returns the next prime number. The most straightforward way of implementing this is to take the number following the previously found prime and to check its primality. If it is not a prime, advance to the following number, and so on. This would require introducing of two new private member variables. The modified Generator.h will look as follows:

Notice how the prime number searching is implemented using the `find_if` algorithm from the C++ Standard Library. This implementation specifies just enough details, deferring the rest of details to the next abstraction layer called PrimeCandidate. We shouldn't concern ourselves about performance at this point, focusing instead on correctness and compactness of the solution. The optimization issues could be addressed later in the development process should the need arise. The new PrimeCandidate.h file looks as follows:

Its structure is very similar to that of Primes.h and includes files specifying particular classes belonging to this layer. The CandidateIterator.h file looks like this:

The PrimeCandidate::Iterator class encapsulates a prime candidate in a form of *iterator* [6]. The initial implementation uses, as candidates, all numbers of the specified type starting from 1. Note that the PrimeCandidate::Iterator class is also parameterized with the type of prime number. The PrimalityTester.h file looks as follows:

By the end of this stage, we have implemented the generator. The generator is implemented in terms of the Iterator that will provide candidates, and in terms of the Tester that will evaluate those candidates for primality. Our next stage will be to implement the Tester.

11 Step 5: Implementing the Tester

The most straightforward implementation of the PrimeCandidate::Tester would be to divide the prime candidate by all numbers smaller than this candidate, checking if at least one of them has a zero remainder. The following modified version of the PrimeCandidate::Tester class *self-operator* implements this algorithm exactly as it sounds:

Note that, like in the case of Primes::Generator, the PrimeCandidate::Tester class is also implemented as a *functor* parameterized with the prime number type. Notice also the role of C++ Standard library elements: `find_if` algorithm, `modulus<T>` *functor*, and `bind1st` and `not1` *helpers* [6], which allow straightforward implementation of validating whether numbers are prime. The usage of the PrimeCandidate::Iterator to enumerate the dividers is intentional, since (unless we are in the optimization phase), the range of dividers we test against, is the same as the range of candidates we are testing.

We now have a fully functional program that will generate prime numbers, and format the output. We now go to the optimization phase

12 Optimization

The current solution works well for generating 500 primes or so (takes less than second). However, it would be quite slow for generating 5000 prime numbers, and for a larger number the performance would be unacceptable.

The good part is that the proposed solution does establish a robust and flexible structure. This structure allows substantial optimization, primarily through the code specialization rather than through code complication.

No less important is the fact that the whole optimization process can be broken into a number of steps, with each step addressing a single specific aspect of the program.

According to the *just-enough improvement* principle of Top-Down Decomposition the next step of optimization would be conducted only if the previous step has been proven to be insufficient and there is no other way to prove it but to measure the performance. For the sake of prime number generator we will use a simple `progress_timer` class supplied as a part of the C++ Boost [2] Timer library [8]. The `progress_timer` class ”automatically measures elapsed time, and then on destruction displays an elapsed time message”. The main program has to be modified as follows:

To improve the Prime Number Generator’s performance, we could modify it in one or both of the following directions:

1. Reduce the number of candidates
2. Reduce the number of dividers

Each optimization will be implemented in one or more separate steps below.

Note 1. This exercise of optimizing the Prime Number Generator is going to emphasize a very important fact about requirements changing. The problem is not that requirements are changing. Had we known from the very beginning, that we would have to calculate all primes within the 32-bit, we could have written the program without applying any specific methodology. The problem is that requirements are constantly changing in unpredictable directions. The market (customers, suppliers, and competitors) do not really care what we did or did not do prepare for in our software. The can and will change the requirements based on their own needs. Under these circumstances the requirements are constantly in various and unpredictable directions. This could be very frustrating for developers that in that it typically will very soon convert what was initially elegant software code into un-maintainable heap. What we are going to demonstrate, is that the Top Down Decomposition approach leads to a very resilient code structure, from the very beginning prepared for a very wide range of changes. This is due to the fact, that at every step we are actually forming a ”family of related programs” [2]. By a certain degree of irony we can build this family because ”certain aspects of the given problem statement are ignored at the beginning” [2].

13 Step 6: Reduce the Number of Candidates

To reduce the number of candidates, we may choose to test only *throdd* numbers.

Definition 2. *Throdd numbers are numbers that are divisible neither by 2, nor by 3 [1].*

The *throdd* number concept is easily encapsulated within the PrimeCandidate::Iterator class, with almost full transparency for the rest of the program. Getting the next *throdd* number is implemented within the increment (++) operator as follows:

The purpose of the nested `fix_vals` type will be explained later. To work properly, the program must treat the numbers 2 and 3 in a special way. There are two possible ways to implement this special treatment:

[2] The C++ Boost Library is a collection of portable C++ libraries, which ”work well with the C++ Standard Library” and ”are suitable for eventual standardization”.

1. Within the operator ++() member function
2. Outside of the PrimeCandidate::Iterator class

The difference is subtle, but important. Implementing a special treatment within the operator ++() seems to be the most straightforward and natural. It also allows hiding all implementation details from higher layer, which seems to be the primary goal of Top Down Decomposition. Sounds OK, right? Well . . . , not exactly.

In order to understand why it's not so straightforward we need to consider the third mental aid mentioned at the beginning of this exercise: enumeration.

13.1 Enumeration

By enumeration we understand analyzing of individual cases to be handled separately. Within the programming languages realm enumeration is implemented using the sequence (semicolon ';' in C++) and `if-then-else` statements. As Dijkstra stated it in [2], enumeration works well only "under the severe boundary condition". In other words the number of special cases to be analyzed must be small. This is especially true for the `if-then-else` statements.

Contrary to the common approach, where these differences are implemented by means of `if-then-else` constructs within inner classes, we propose raising them to the topmost level. In many cases that would allow replacing the `if-then-else` statements with a sequence. This in turn allows such implementation of the inner classes that relays solely on pre-conditions and avoids paranoiac repetitive checks of boundary conditions. To sum, rising the constructs to the topmost level reduces complexity and improves performance.

To a certain degree this approach was reflected in [12] where Saltzer, Reed and Clark formulated so-called *end-to-end argument in system design* postulating that: ". . . functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level". By *cost* we have to understand not only computer resources such as memory or CPU, but also the overall system complexity.

Note 2. This may seem to be a contradiction to the arguments made above in favor of abstraction. It is not. The major purpose of abstraction is to combat the complexity that arises from enumeration. If the enumeration occurs at a certain layer it can be dealt with at this layer - typically though introduction of a new abstraction layer underneath - and/or may be propagated through the higher layers in order to fine-tune the abstraction specifications. The very purpose of this process is to come up with as thin as possible an interface between two subsequent layers in order to keep each of them as simple as possible. In our case propagation of special treatment of 2 and 3 to the upper layer does not make its *programming* significantly more complex, but it does simplify the PrimeCandidate::Iterator *computation* substantially [3].

To achieve that goal a special version of `generate_n` algorithm is created as follows:

[3] See [2] for the difference between *programming* and *computation*

This is a special version of `generate_n` algorithm, and it acts as interface *glue* between the main program and the Primes layer. This special version of `generate_n` algorithm is implemented using C++ function overloading for Primes::Generator class and for that reason the main loop did not need to change at all. The Primes.h file is changed to include the new header file as follows:

The special values 2 and 3 treatment is implemented using the C++ Boost Meta-Programming Library (MPL) [9], which ”is a C++ template metaprogramming framework of compile-time algorithms, sequences and metafunction classes”.

In implementing the special version of `generate_n` algorithm we assume that the overall number of fixed values is small and thus we do not want to handle them in a run-time loop, but rather prefer to generate a corresponding number of lines of code, which will send these fixed values to the output iterator one-by-one. It’s for that purpose the MPL’s version of the `for_each` algorithm does exist. It is similar to the C++ Standard Library `for_each`, but is evaluated during the compile time. In that specific case it will generate invocations of the `copy_one` inline function that will copy exactly one fixed value to the output iterator. To properly pass the output iterator and counter by reference we use the C++ Boost’s `bind` [10] and `ref` [11] template functions. The `bind` template function is ”a generalization of the standard functions `std::bind1st` and `std::bind2nd`”. The `ref` template function allows the ”passing references to function templates (algorithms) that would usually take copies of their arguments”.

Once we’ve finished with the fixed values special treatment all what we need is to invoke the C++ Standard Library version of `generate_n`. In order to achieve a proper type resolution to prevent recursive calls we again use the C++ Boost’s `bind` template function.

The fixed values are organized in a compile-time list using the C++ Boost MPL’s `list_c` container [9]. These fixed values naturally belong to the PrimeCandidate::Iterator class and are defined as a nested type (see above). The Primes::Generator class has just to export these fixed values to make them available for the `generate_n` algorithm. The new version of Primes::Generator class looks as follows:

At this point we optimized our candidate iterator so that it does not supply numbers that are devisable by 2 or by 3.

14 Step 7: Reduce the Number of Dividers

The optimization applied in the previous step makes the prime number generator perform faster (about 75%).This suffices for generating 5000 primes, but is far from being fast enough for generating say 10000 primes. In order to achieve even better performance we have to significantly reduce the number of dividers.

In order to establish the primality of a *throdd* p, we do not really need to test p against ALL the *throdss* smaller then p, it is sufficient that we test against all *throdds* that are smaller then $\sqrt{p}$. If the candidate p does not have any *throdd* devisors smaller then $\sqrt{p}$ it is a prime [1].

The new version of the Tester *functor* will look like this:

This simple change offers a significant improvement in run time. Instead of taking $O(n)$ to test the primality of n, our program will take $O(\sqrt{n})$. This change was totally confined to the Tester function.

It is important to note that at this point the generator has to treat the tester as a statefull object, which requires the following modification of the Primes::Generator class:

Note, that we ensure the statefullness of the tester through a combination of the C++ Boost's `bind` and `ref` templates [10, 11].

15 Step 8: Use Stored Composites

To even farther reduce the number of dividers we test against, we will store a list of composite numbers obtained as product of previously generated prime numbers [2].

Note 3. Using the composite numbers eliminates the need for expensive modulus operation. See [2] for more details.

For each previously generated prime number p, its first composite is stored in a form of $m = p^2$. To check if a candidate k is a prime, we compare it with each composite m_i created from a prime p_i, such that $p_i \leq \sqrt{k}$. If the composite m_i is less than k, it should be increased. How much it should be increased by poses an interesting problem, which we will deal with later.

According to the *Linear Search Theorem* [2] we could stop the process once we've found the first composite, which is greater then k. This leads us to an interesting modification of the PrimeCandidate::Tester class as follows:

This new implementation eliminates the `find_if` linear search algorithm completely and uses the `push_heap` and `pop_heap` algorithms [6]. Now instead of taking $O(\sqrt{n})$ to test the primality of n, our program will take $O(\log \sqrt{n})$, which is substantially faster. We store the composites in the `vector` container supplied by the C++ Standard Library [6].

Introducing the composite number concept requires a new abstraction layer, which fits naturally into the existing infrastructure. The Composite.h file looks as follows:

The Primes::Number::Composite class stores the current value of a composite and its next increment. For each prime p its corresponding increment could have been p, but that would have generated even composites, which is wasteful. We could have used $2 \cdot p$ as our increment, but that would have generated multiples of 3. Ideally the composite's increment should change in phase with the corresponding prime *throdd* it was initially created from.

Example 1. For the prime number 5 the next *throdds* are obtained using 2, 4, 2, . . . as increments in that order leading to 7, 11, 13, ... For the prime number 7 the next *throdds* are obtained using 4, 2, 4, . . . as increments in that order leading to 11, 13, 17, . . . correspondingly. Accordingly from the prime number 5 the composites will be generated starting from $5^2 = 25$ using 10, 20, 10, . . . as increments, while from the prime number 7 the composites will be generated starting from $7^2 = 49$ using 28, 14, 28, . . . as increments correspondingly.

This leads us to the conclusion that the composite's increment is obtained from its corresponding prime, which justifies the introduction of a new abstraction called CandidateValue as follows:

The Primes::Number::Candidate class by itself is built on top of a lower level abstraction called Increment, which is defined as follows:

Here we specify two abstractions. The first abstraction is called Primes:: Increment::Add and is intended for getting the next prime candidate. The second abstraction is called Primes::Increment::Multiply and is intended for getting the next composite number. The both abstractions are tightly coupled together and thus are defined in the same header file [4].

Note that the fixed values concept migrated to the Primes::Increment layer. All arithmetic calculations are performed during compile time using the C++ Boost MPL library [9]. In particular the `mpl::fold` template is used for obtaining a product of fixed values in compile time. For more details see [9].

Note, that with the exception of a change in the namespace structure there is no need to modify the Primes::Generator class since the Primes:: Candidate::Tester class statefullness has already been ensured in the previous step.

The last optimization allows the prime generator to perform extremely fast and makes it suitable for generation of a really large number of primes.

16 Step 9: Calculate All 32-bit Primes

In order to generate all primes within the 32-bit range we need to deal with memory rather than CPU time optimization. Keeping composite numbers for all primes will put a huge demand on the cache-memory size. Even if such amount of memory is available it will eventually slow down the algorithm significantly.

The good news is that there is no need to store all composite numbers, but rather only those, which were created from primes that are less or equal to the square root of the last prime number p_N.

According to [13] for each $N \geq 6$ the N^{th} prime number P_N can be bound as

$$N \cdot \ln N < P_N < N \cdot (\ln N + \ln \ln N) \tag{1}$$

On the other hand, according to [13] for each $x \geq 17$

$$\pi(x) > \frac{x}{\ln x} \tag{2}$$

and for $x > 1$

$$\pi(x) < 1.25506 \cdot \frac{x}{\ln x} \tag{3}$$

,where $\pi(x)$ is the number of primes $\leq x$. Armed with these formulas we can finally modify our program to accomplish the task of generating all primes within the 32-bit range. The number of changes will be negligibly small compared with the challenge.

[4] In this implementation we assume that bit test,bit shift and bit inversion operations are cheaper than a generic *add* operation, which is usually true for typical binary processors such as Intel's Pentium.

First of all for debugging and development speed process we do not want to deal with 32-bit range until the very last stage when we are confident with the program correctness. A more generalized problem statement would sound like ”generate all prime numbers for the given type T”. Since all layers are already templetized with the prime data type, all that is required is to modify the `main` function such that it will obtain the actual type as a macro from compiler command line and to use `unsigned short` for debugging purposes.

We also need to modify the `main` function to correctly calculate the column width. The new version will look as follows:

As we can see the actual work of the column width calculation is delegated to the Primes::TabOutIterator class, which has to be modified as follows:

This in turn requires an introduction of a new `NthPrime` template function as follows:

The customized version of the `generate_n` algorithm has to be modified as follows:

The new version of the `generate_n` algorithm now operates in three phases:

1. Output the fixed values.
2. Generate enough primes to populate the cache of composites.
3. Generate the remainder of the primes, using the cache.

In order to estimate the number of composites to be stored a new `CacheSize` function is introduced as follows:

The Primes::Generator class is modified and now provides two versions of the *self-operator*: one, which will store composites in the cache, and one, which will just use it, but will handle properly the candidate overflow race condition. The new version of the Generator.h file looks as follows:

In this version of Primes::Generator the handling of overflow race condition relies on the fact that the first after overflow (end) value of prime candidate is equal to its first value, namely 1. To prove this assumption we have to prove that for any $N > 0$: $4^N - 1$ is not a *throdd*, and $4^N - 3$ is a *throdd* [5].

Proof. To prove the first statement we have to prove that for any $N > 0$: $4^N - 1$ is divisible by either 2 or 3. It’s obviously not divisible by 2, but it is divisible by 3, which can be proven by induction as follows:

1. for $N = 1$: $4 - 1 = 3$
2. for $N > 1$: $4^N - 1 = 3 \cdot 4^{(N-1)} + (4^{(N-1)} - 1)$

To prove the second statement we have to prove that for any $N > 0$: $4^N - 3$ is not divisible by neither 2, nor 3, which is obvious. □

Note 4. This limited arithmetic exercise demonstrates a very common trade-off between the three types of effort we must always deal with :intellectual, programming and computational. In this particular case, had we not been willing to pay the price of the intellectual effort needed to prove that a prime candidate’s end value is equal to its

[5] And thus adding 4 to this number will lead to 1

first value we would have either been forced to litter the code with the explicit computation of the end value or to pay additional CPU cycles on checking if the next candidate value is less than the previous value. Being aware about his trade - off is extremely important for making the software development work effective.

The last modification, which is required, is to make public the `Store` method of the Primes::Candidate::Tester class in order to allow the Primes::Generator to access it.

17 Summary

The primary goal of Top-Down Decomposition is controlling complexity. Others goals, such as efficiency and reuse, are of secondary importance and are fully subordinated to the primary goal. To achieve the required control over complexity, we employ abstraction. We have shown how, using the C++ Standard Library algorithms we can effectively abstract the most of the typical iterative processes. When enumerating or doing special case analysis, pushing the functions to the topmost possible level, yields more efficient and robust code.

It takes a Pentium-III computer about 2.5 hours to generate all primes within the 32-bit range. The largest 32-bit range prime number is 4294967291. Having all layers parameterized with the prime number type allows significant reduction of the debugging time, through temporal usage of the `unsigned short` type.

Bibliography

[1] Dijkstra, E.W.: ”Stepwise program construction”, http://www.cs.utexas.edu/ users/ewd/ewd02xx/ewd227.pdf, February 1968

[2] Dijkstra, E.W.: ”Notes on Structured Programming”, http://www.cs.utexas.edu/ users/ewd/ewd02xx/ewd249.pdf, April 1970

[3] Buschman, F., et al: ”Pattern-Oriented Software Architecture”, John Wiley & Sons, 1996

[4] Czarnecki K., Eisenecker U.W.: ”Generative Programming: Methods, Tools and Applications”, Addison-Wesley, 2000

[5] Coplien, J.: ”Multi-Paradigm Design for C++”, Addison-Wesley, 1999

[6] Dinkum C++ Library Reference, http://www.dinkumware.com/manuals

[7] Martin, Robert C.: ”Agile Software Development, Principles, Patterns, and Practices”, Prentice Hall, 2002

[8] C++ Boost Timer Library: http:://www.boost.org/libs/timer/timer.htm

[9] C++ Boost MPL Library: http:://www.boost.org/libs/mpl/doc/index.htm

[10] C++ Boost Bind Library: http:://www.boost.org/libs/bind/bind.html

[11] C++ Boost Ref Library: http:://www.boost.org/doc/html/ref.html

[12] Saltzer, J.H., Reed, D.P., Clark, D.D.: ”End-to-end Arguments in System Design”, ACM Transactions on Computer Systems 2, 4, November 1984

[13] Menezes, A.J., Van Oorschot, P.C., Vanstone A.A.: ”Handbook of Applied Cryptography”, CRC Press, 1996

SOUL and Smalltalk - Just Married

Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis

Kris Gybels[*]

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Elsene, Belgium
kris.gybels@vub.ac.be

1 Introduction

The *Smalltalk Open Unification Language* is a Prolog-like language embedded in the object-oriented language Smalltalk [5]. Over the years, it has been used as a research platform for applying logic programming to a variety of problems in object-oriented software engineering, some examples are: representing domain knowledge explicitly [3]; reasoning about object-oriented design [15, 14]; checking and enforcing programming patterns [11]; ; checking architectural conformance [16] and making the crosscuts in Aspect-Oriented Programming more robust [6]. These examples fit in the wider research of *Declarative Meta Programming*, where SOUL is used as a meta language to reason about Smalltalk *code*.

Recently, we explored a different usage of SOUL in connecting business rules and core application functionality [2], which involves reasoning about Smalltalk *objects*. We found we had to improve on SOUL's existing mechanism for interacting with those objects because it was not transparent: it was clear from the SOUL code when rules were invoked and when messages were sent to objects, vice-versa solving queries from methods was rather clumsy. Ideally we would like to achieve a *linguistic symbiosis* between the two languages: the possibility for programs to call programs written in another language as if they were written in the same [8, 13]. Such a transparent interaction would make it easy to selectively change the paradigm parts of an application are written in: if we find that a Smalltalk method is better written as a logic rule we should be able to replace it as such without having to change all messages invoking that method.

We will here take a historical approach to describing the SOUL/Smalltalk symbiosis. We would like to provide an insight into our motivation for and approach to achieve the symbiosis by contrasting three distinct stages in its evolution. In a first stage, SOUL was developed as a direct Prolog-derivate with some additional mechanisms for manipulating Smalltalk objects as Prolog values. In a second and third stage we explored alternative mechanisms and a more Smalltalk-fitting syntax for SOUL. Interestingly, when we performed a survey of other combinations of object-oriented and logic programming we found we could easily categorize their approaches into one of our three

[*] Research assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

”stages”. The following sections discuss the stages in detail and the ”Related Work” section at the end briefly discusses the survey.

2 Stage 1: Escaping from SOUL

The interaction mechanism found in the original SOUL can best be characterized as an escape mechanism. But before we go into this, let us make some general points about this version of SOUL:

Implementation: SOUL is embedded in Smalltalk, meaning it is entirely implemented in it.

Syntax: We assume readers are familiar with Prolog, the differences with this language and SOUL in this stage are:

- **Variable notation:** in Prolog, variables are written as names starting with a capital letter, in SOUL they are written as names preceded with a question mark, thus `Something` translates to `?something`.
- **List notation:** in Prolog, square brackets (`[ ]`) are used to write lists, these are replaced with angular brackets in SOUL (`< >`).
- **Rule notation:** the ’if’ operator `:-` linking conclusion to conditions is replaced with the `if` operator in SOUL.

The combination of Smalltalk’s Meta-Object Protocol and SOUL’s embedding in Smalltalk lead to the insight that the simplest way to let SOUL programs reason about Smalltalk code is to give them access to the meta-objects directly. For this reason there are additional differences with Prolog:

Values: any Smalltalk object (not just the meta-objects) can be bound as a value to a logic variable.

Syntax: the Smalltalk term, a snippet of Smalltalk code enclosed in square brackets `[ ]`. The Smalltalk code can contain logic variables wherever Smalltalk variables are allowed.

Semantics: when Smalltalk terms are encountered as conditions in rules, they are ”proven” by executing the Smalltalk code. The return value should be a boolean, which is interpreted as success or failure of the ”proof”. Smalltalk terms can also be used as arguments to conditions, then they are evaluated and the resulting value is used as the value of the argument. Unification deals with Smalltalk objects as follows: two references to an object unify only if they refer to the same object.

Primitive predicates: a primitive predicate `generate` can be used to generate elements of a Smalltalk collection as successive solutions for a variable.

The example set of rules in figure 1 are taken from SOUL’s library for Declarative Meta Programming and show how Smalltalk terms are used. A predicate `class` is defined which reifies class meta-objects into SOUL; two different rules are defined for it to deal efficiently with different argument binding patterns. The `subclass` predicate expresses that two classes are related by a direct subclassing relationship when one is

```
class(?x) if
  var(?x),
  generate(?x, [ System allClasses ])
class(?x) if
  nonvar(?x),
  [ ?x isClass ]

subclass(?super, ?sub) if
  class(?sub),
  equals(?super, [ ?sub superclass ])

hierarchy(?root, ?child) if
  subclass(?root, ?child)
hierarchy(?root, ?child) if
  subclass(?root, ?direct),
  hierarchy(?direct, ?child)
```

Fig. 1. Example rules defining predicates for reasoning about Smalltalk programs.

```
argumentArray := Array with: (Array with: #x with: someClass).
evaluator := SOULEvaluator eval: ’if hierarchy(?x, ?y)’
                           withArgs: argumentArray.
results := evaluator allResults.
ysolutions := results bindingsForVariableNamed: #y.
```

Fig. 2. Code illustrating how the SOUL evaluator is called from Smalltalk and how the results are retrieved.

the answer to the `subclass` message sent to the other. The `hierarchy` predicate extends this to indirect subclassing relationships.

The example rules are indicative for the way SOUL interacts with Smalltalk in this stage: the use of Smalltalk terms is limited to a small collection of predicates such as `class` and `subclass`, which are organized in the so-called ”basic layer”. Other, more high-level predicates such as `hierarchy` make use of the predicates in the basic layer to interact with Smalltalk objects. This organization avoids pollution of the higher-layer predicates with explicit escape code[1]. In a way, the basic layer provides a gateway between the two languages by translating messages to predicates and vice-versa.

The other direction of interaction, from Smalltalk to SOUL, is done through explicit calling of the SOUL evaluator with the query to be evaluated passed as a string. Figure 2 illustrates how the `hierarchy` predicate is to be called. On the second line, an evaluator object is created by sending the message `eval:withArgs:` to the `SOULEvaluator` class, the message is passed the query to evaluate and variable bindings as arguments. The variable bindings are passed as an array of variable-

[1] Another reason why this is done is to make the higher-layer predicates less dependent on Smalltalk, so that they may later be used when reasoning about code in other OO languages [4].

object pairs. In the example, the logic variable `?x` will be bound to the value of the Smalltalk variable `someclass`, so the query will search for all child classes of that class. These child classes will then be bound as solutions to the variable `?y`. These solutions can be retrieved by sending an `allResults` message to the evaluator object, which returns a result object. The result object then needs to be sent the message `bindingsForVariableNamed` to actually retrieve the bindings, which are returned as a collection.

3 Stage 2: Predicates as Messages

A second stage of SOUL-Smalltalk interaction, which we reported on at a previous multi-paradigm programming workshop [1], aimed at providing more of a transparent interaction. Our motivation then was especially to improve on the way Smalltalk programs can invoke queries, and do it in a way that would provide *linguistic symbiosis*. To do so, we tried to map invocation of predicates more directly to the concept of sending a message.

The term linguistic symbiosis refers to the ability for programs to call programs written in another language as if they were written in the same. Having this ability would also imply that transparent replacement is possible: replacing a ”procedure” (= procedure/function/method/...) in the one language with a ”procedure” in the other, without having to change the other parts of the program that make use of that ”procedure”. In fact, the term was coined in the work of Ichisugi et al. on an interpreter written in C++ which could have all of its parts replaced with parts written in the language it interprets. Such usage of linguistic symbiosis to provide reflection was further explored in the work of Steyaert [13].

While these earlier works provided us with solutions, we also had an added problem: the earlier works dealt with combining two languages founded on the object-oriented paradigm, while we aimed at combining an object-oriented and a logic language. The earlier works dealt with mapping a message in the one language to a message in the other, while we needed to map messages to queries.

To provide a mapping of messages and queries, we had five issues to resolve:

Unbound variables: how does one specify in such a message that some arguments are to be left unbound? The concept of ’unbound variables’ is foreign to Smalltalk.

Predicate name: how is the name of the predicate to invoke derived from the name of the message?

Returning multiple variables: how will the solutions be returned when there are multiple variables in the query?

Returning multiple bindings: if there are multiple solutions for a variable, how will these be returned?

Receiver: which object will the message be sent to?

We combined the solution for the first two issues by assuming that predicate names, like Smalltalk messages, would be composed of keywords, one for each argument. To specify which variables to leave unbound we adopted a scheme for combining these keywords into a message name from which that specification can be derived. To invoke a

Message	Query
Main add: 1 with: 2 to: 3	if Main.add:with:to:(1,2,3)
Main add: 1 with: 2	if Main.add:with:to:(1,2,?res)
Main add: 1	if Main.add:with:to:(1,?y,?res)
Main add	if Main.add:with:to:(?x,?y,?res)
Main addwith: 2	if Main.add:with:to:(?x,2,?res)
Main addwithto: 3	if Main.add:with:to:(?x,?y,3)
Main addwith: 2 to: 3	if Main.add:with:to:(?x,2,3)
Main add: 1 withto: 3	if Main.add:with:to(1,?y,3)

Table 1. Mapping a predicate to messages

predicate from Smalltalk one would write the message as: the name of the first keyword, optionally followed by a colon if the first argument is to be bound and a Smalltalk expression for the argument's value, then the second keyword, concatenated to the first if that one was not followed by a colon, and again itself followed by a colon if needed for an argument and so on for the other keywords until no more keywords need to follow which take an argument. This is best illustrated with an example. Table 1 shows the 2^3 ways of invoking a predicate called `add:with:to:` and the equivalent query in SOUL.

For the issue of needing a receiver object for the message, we mapped layers to objects stored in global variables. Because in Smalltalk classes are also objects stored in global variables, this has the effect of making a predicate-invoking message seem like a message to a class. The basic layer is for example stored in `Basic`.

We proposed two alternative solutions to the issues of returning bindings. The first was simply to return as result of the message a collection of collections: a collection containing for each variable a collection of all the bindings for that variable. The alternative consisted of returning a collection of message forwarding objects, one for each variable. Sending a message to such a forwarding object would make it send the same message to all the objects bound to the variable. The idea was to provide an implicit mechanism for iterating over all the solutions of a variable, very much how like SOUL can backtrack to loop over all the solutions for a condition. This however lead to matters such as whether forwarding objects should also start backtracking over solutions etc., so it was discarded as a viable solution. We coined the term *paradigm leak* to refer to this problem of concepts ”leaking” from one paradigm to the other.

We also used the predicate and message mapping to replace SOUL's earlier use of Smalltalk terms. Instead of using square brackets to escape to Smalltalk for sending a message, the same message can now be written more implicitly as an invocation of a predicate in an object ”pretending to be a SOUL module”. Here, the reverse of the above translation happens: SOUL will transform the predicate to a Smalltalk message by associating the arguments of the predicate to the keywords in its name. The predicate's last argument will be unified with the result of the actual message send. Take the following example:

```
if Array.with:with:with:(10,20,30, ?instance),
   ?instance.at:(2,?value)
```

```
member(?x, <?x | ?rest>).
member(?x, <?y | ?rest>) if
  member(?x, ?rest).

<?x | ?rest> contains: ?x.
<?y | ?rest> contains: ?x if
  ?rest contains: ?x.
```

Fig. 3. Comparison of list-containment predicate in classic and new SOUL syntax.

The first condition in the example query will actually be evaluated by sending the message `with: 10 with: 20 with: 30` to the class `Array`. The result of that message is a new `Array` instance, which will be bound to the variable `?instance`. In the second condition, the message `at: 2` will be sent to the instance and the result, 20 in this case, will be bound to the variable `?value`.

While in this second-stage SOUL mixing methods and rules is entirely transparent from a technical standpoint, it is obvious which code is intended to invoke what to a human interpreter. Technically there is no more need in SOUL for an escape mechanism, and the same language construct is used to invoke rules and messages. Similarly in Smalltalk, queries no longer have to be put into strings to let them escape to SOUL and can just be written as message sends. However, a Smalltalk programmer would frown when seeing messages such as `addwith: 2 to: 3`. Furthermore, he would probably guess that the result of that message would be the value 5, instead it will be a collection with a collection containing the value 5. The keyword-concatenated predicate names in SOUL also lead to awkward looking programs in that language.

4 Stage 3: Linguistic Symbiosis?

The next, and currently last, stage in the SOUL-Smalltalk symbiosis uses a new syntax for SOUL to avoid the clumsy name mappings from the previous stage. For this stage we also had a specific application for the symbiosis in mind, business rules [2], which influenced its development in certain respects. One difference is that previously we wanted to allow Smalltalk programs to call the existing library of SOUL code-reasoning predicates, while for supporting business applications the idea is rather to use SOUL to write new rules implementing so-called business rules of the application. This also implies another shift: reasoning about (business) objects rather than meta objects.

In the new syntax predicates look like message sends. Let us illustrate with an example, figure 3 contrasts the classic `member` predicate with its new `contains:` counterpart.

The second rule for `contains:` can be read declaratively simply in Prolog-style as ”for all `?x`, `?y` and `?rest` the `contains:` predicate over `<?y | ?rest>` and `?x` holds if ....”. A declarative message-like interpretation could read ”for all `?x`, the answer to the message `contains: ?x` of objects matching `<?y | ?rest>` is true if the answer of the object `?rest` to `contains: ?x` is true.” Both interpretations are equivalent, though the second one is really the basis for the new symbiosis.

```
?product discountFor: ?customer = 10 if
  ?customer loyaltyRating = ?rating &
  ?rating isHighRating
```

Fig. 4. Example of a rule using the equality operator.

Because messages can return values other than booleans, we added another syntactic element to SOUL to translate this concept to logic programming. The equality sign is used to explicitly state that ”the answer to the message on the left hand side of `=` is the value on the right hand side”. Figure 4 shows an example.

The new syntax has a two-fold impact on how the switching between Smalltalk and SOUL occurs. It is no longer necessary to employ a complicated scheme with concatenation of keywords to get the name of a predicate. Another is that there is no more mapping of objects to SOUL modules and vice-versa, modules were dropped from SOUL as the concept of having a ”receiver” for a predicate now comes as part of the message syntax.

A Smalltalk program no longer has to send a message to a SOUL module ”pretending to be an object” to invoke a query. Instead, a switch between the two languages now occurs as an effect of method and rule lookup: we changed Smalltalk so that when a message is sent to an object and that object has no method for it, the message is translated to a query. In SOUL, when a rule is not found for the predicate of a condition, the condition is translated to a message. This new scheme makes it much easier and much more transparent to actually interchange methods and rules.

The translation of queries and messages is straightforward and we'll simply illustrate with another example. Figure 5 shows a price calculation method on a class `Purchase` which loops through all products a customer bought and sums up their total price minus a certain discount. When the `discountFor: customer` message is sent to the products, Smalltalk will find no method for that message, so it will be translated to the query:

```
if ?arg1 discountFor: ?arg2 = ?result
```

Where `?arg1` and `?arg2` are already bound to the objects that were passed as arguments to the message. When the query is finished, the object in `?result` is returned as result of the ”message”. This returning of results is actually a bit more involved, we'll discuss it further in the next section.

For the inverse interaction, we can take the `loyaltyRating =` condition in the `discountFor:` rule (fig. 4) as an example. For a small business the loyalty rating of a customer can simply be stored as a property of the customer object which can be accessed through the `loyaltyRating` message. In that case, SOUL will find no rule for the ”predicate” `loyaltyRating` and will translate the condition simply to the message `loyaltyRating` which is then sent to the customer object in the variable `?customer`. After it returns, the result of the message is unified with the variable `?rating`. Of course, for a bigger business we might want to replace the calculation of `loyaltyRating` with a set of more involved business rules which we'd prefer to

```
Purchase instanceVariables: ’shoppingBasket customer’

Purchase>>totalPrice

  | totalPrice discountFactor |

  totalPrice := 0.
  shoppingBasketContents do: [ :aProduct |
    discountedFactor :=
        (100 - (aProduct discountFor: customer)) / 100.
    totalPrice :=
        totalPrice + (discountFactor * aProduct price).
  ]
```

Fig. 5. Example price calculation method on Purchase class

implement with logic programming, for example ”a high rating is given to a customer when she has already spent a lot in the past few months”. With the transparent symbiosis such a replacement is easy to do.

5 Limits and Issues

At the end of the ”Stage 2” section, we remarked that our solution then was only technically transparent, it was rather obvious to a programmer which code was intended to invoke which paradigm. In the previous section we demonstrated that this is now much less the case, it is fairly easy now to interchange methods and rules without this becoming obvious. There are however limits to this interchanging and there are still subtle hints that may reveal what paradigm is invoked. These limits and issues stem from differences in programming style between the object-oriented and logic paradigms.

One important style difference between the paradigms is the way multiplicity is dealt with. In logic programming, there is no difference between using a predicate that has only one solution and one that has multiple solutions. In object-oriented programming there is an explicit difference between having a message return a single object or a collection of objects (even, or especially, if there’s only one object in that collection). This difference leads to an issue in how results are returned from queries to Smalltalk, and one in how predicates and messages are named.

When a Smalltalk message invokes a SOUL query and the query has only one solution, should the solution object be simply returned or should a singleton collection with that object be returned? The invoking method may expect a collection of objects, which would then just happen to contain just a single item, or it may generally be expecting there to be only one result. It is difficult for SOUL however to know which is the case. To deal with this we made SOUL return single solutions in a `FakeSingleItemCollection` wrapper. The `FakeSingleItemCollection` class implements most of the messages expected of collections in Smalltalk, any other

```
?child ancestor = ?parent if
  ?child parent = ?parent.
?person ancestor = ?ancestor if
  ?person parent = ?parent,
  ?parent ancestor = ?ancestor
```

Fig. 6. Rules expressing the ancestor relationship between Persons

```
Person instanceVariables: ’name parent’

Person>>parent
   ˆ parent

Person>>name
  ˆ name

Person>>printOn: stream

  name , ’ descendant of ’ printOn: stream.
  self ancestor do: [ :ancestor |
     ancestor name , ’ and ’ printOn: stream
  ]
```

Fig. 7. Instance variables and some methods of the Person class

messages are forwarded to the object that is being wrapped. There is thus an ”automatic adaptation” to the expectations of the invoking method.

Plurality, or lack thereof, in the names of predicates and messages can cause some programming style difficulties. Figures 6 and 7 illustrate the modeling of persons and their ancestral relations through a class and some logic rules. Invoking these rules from the `printOn:` method is however awkward: it is quite natural for a logic programmer to write the relationship as ”ancestor” even though there will be multiple ancestors for each Person, the object-oriented programmer would however prefer to write the plural ”ancestors” to indicate that a collection of results is expected. One solution to this problem is to implement a rule for `ancestors` which simply maps to `ancestor`, this would however defeat the purpose of having an automatic mapping of messages and queries. A potential solution could be to take this style difference into account when doing the mapping by adding or removing the suffix *-s* when needed.

When comparing the stage 2 and stage 3 symbiosis, stage 3 may seem more limited in the variables that can be left unbound when invoking queries from Smalltalk. In stage 2 the mapping of predicate names to message names implicitly also indicated which variables to leave unbound, while in stage 3 the mapping of messages to queries only leaves unbound the result variable, the one on the right hand side of the equality sign in the query. Actually, we did implement a means for leaving other variables unbound as well. We changed the way Smalltalk deals with temporary variables to allow for the following code to be written:

```
| products customers discounts |

discounts := products discountFor: customers
```

Normally the Smalltalk development environment would warn that this code uses temporary variables before they are assigned. Now however, the message `products discountFor: customers` will result in the query:

```
if ?arg1 discountFor: ?arg2 = ?result
```

Where all of `?arg1`, `?arg2` and `?result` are left unbound. When the query is finished, the result of the message will be as described earlier and additionally the temporary variables `products` and `customers` will also be assigned the solutions of the variables `?arg1` and `?arg2`.

This leaving unbound of temporary variables is however another example of a paradigm leak, it is quite unnatural code for a Smalltalk programmer to write. We consider it as something that should be used with care and preferably avoided. While in stage 2 the equivalent mechanism seemed most necessary because the motivation was to allow access to all *existing* SOUL predicates, our focus shift to implementing *new* business rules makes it less necessary: its better to design the rules differently. Nevertheless many of the rules will be designed to be used from other rules, not to be replaced with methods and callable in a multi-way fashion. On occasion, these rules may need to be used directly from a method, so we kept the unbound temporaries mechanism in place.

There is also a limitation in leaving arguments unbound the other way around: when translating a condition to a message, all of its arguments are expected to be bound. SOUL will currently generate an error otherwise. It would be possible though to at least deal with the ”receiver” argument of the condition in a more logic-like way: when it is unbound, SOUL could send the message to some random object from memory which support a method for the message. If the message's result is true, the object is a solution for the ”receiver” variable. On backtracking all of the other objects supporting the message would be tried. A problem here would be the accidental invocation of object-mutating messages due to polymorphism: when posing the query `?game draw` we may simply be interested in all chess games that ended in a draw but may wind up also drawing all graphical objects on the screen. In practice though this may not be so much of a problem as normally the messages invoked from SOUL would have a keyword ”is” or ”has” in their name because they are written as invocation of predicates, and it is a convention normally applied by Smalltalk programmers as well.

6 Related work

We examined several existing systems which were designed or could be used for business rule development and in which object-oriented and logic programming are combined [2]. The interaction mechanisms we encountered fit in one of three categories similar to the three stages we discussed here: use of an escape mechanism, some explicit mapping of predicates and methods or a syntactic and semantic integration of the two languages. We limit our discussion here mostly to a few systems that aim for the third category as well.

NéOpus also extends Smalltalk with logic programming, though with production-rule based logic rather than proof-based logic [12]. Rules consist of conditions and actions, rather than conclusions, which are respectively expressed as boolean messages to objects and state-changing messages to objects. The concept of a ”conclusion” as something separate from a direct effect on the state of objects is thus dropped. Rules are also not invoked through queries, but rather are triggered by changes in the state of objects and there is no backtracking to generate multiple solutions. This means that some of the issues we had to deal with do not occur in NéOpus: the problems of mapping predicates and methods, returning of multiple results etc. Pachet in fact argues against adding backward chained inferencing to NéOpus because he finds there's a contradiction between the desire to use the OO language to express rules in and allow backward chaining [12], which may come down to our issues. Note that we made the rule language resemble the object-oriented one as closely as possible and needed to allow for symbiosis, we did not simply use the OO language directly to express rules. Besides what form of chaining to use, Pachet also discusses other questions which we had to resolve as well. Most importantly what happens to pattern matching and object encapsulation. Often in logic programming a data structure is accessed directly through unification of its constituent parts. In some of the other systems we examined, like CommonRules [7] , this is still done this way by mapping objects to predicates with an argument for each instance variable. In SOUL, as in NéOpus, we chose to uphold object encapsulation and only allow accessing objects through message sending.

LaLonde and Van Gulik used Smalltalk's reflection to turn ordinary methods into backtracking methods [10]. They built a small framework[2] to support the backtracking methods. Most important in there is a message which makes its calling method return with a certain value but remembers the calling point, the method can then be made to resume execution from that point on. This is achieved by exploiting Smalltalk's ability to access the execution stack from any method. The backtracking takes care of undoing changes to local variables, though not to instance variables and globals . Local variables are thus used to simulate logic variables, but they are assigned rather than unified and there is no simulation like our unbound temporaries for calling methods with some arguments left unbound, so backtracking methods are no full simulation of logic rules. Despite the similarities in the use of Smalltalk expressions, programming in this system seems quite different from programming in symbiotic SOUL.

Kiev [9] extends Java with logic rules, which can be added directly to classes and called through message sending. To call a rule with unbound arguments, one passes an empty wrapper object as argument which will then be bound by the rule. A new for-construct can be used to iterate over all solutions. There is no equivalent for our equality operator construct, calling a rule from a method as a message always returns a boolean to indicate success or failure. This is a subtle but important difference with

[2] Small enough to have the full code listed in their paper.

symbiotic SOUL: returning objects from rules requires the use of sending a message with unbound arguments, making calling rules not as transparent.

7 Summary and Conclusions

We presented the history of a combination of Smalltalk with a logic language. Three distinct stages appeared in its evolution of the interaction between the two languages which we also encountered in studying other combinations of object-oriented and logic programming: a stage where the languages could bind each other's values to variables and manipulate these values by ”escaping” to the other language, a stage where the escape mechanism was made more transparent by an automatic mapping of predicates and methods and the current final stage in which the syntax of the logic language has been adapted to that of the host language to allow not only for technical but programming style transparency as well. The aim was to achieve a linguistic symbiosis so that methods and rules can be easily and transparently interchanged. This is not just of theoretical interest but has an application in the development of business rule applications: an existing application without business rule separation may need to be turned into one that does, or new developments in the policies of the business may make it more interesting to turn methods into rules.

We compared our earlier and current solution for such issues as how to map messages and queries, return multiple results from a query to Smalltalk etc. There are unfortunately still some minor issues to resolve such as how to deal properly with the difference in use of plurality in names between the two paradigms and avoiding the invocation of state-changing messages. Nevertheless we have found the current version of symbiotic SOUL to be a great improvement over previous versions.

Bibliography

[1] Johan Brichau, Kris Gybels, and Roel Wuyts. Towards a linguistic symbiosis of an object-oriented and logic programming language. In Jörg Striegnitz, Kei Davis, and Yannis Smaragdakis, editors, *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2002.

[2] Maja D'Hondt and Kris Gybels. Linguistic symbiosis for the automatic connection of business rules and object-oriented application functionality. (to appear), 2003.

[3] Maja D'Hondt, Wolfgang De Meuter, and Roel Wuyts. Using reflective logic programming to describe domain knowledge as an aspect. In *First Symposium on Generative and Component-Based Software Engineering*, 1999.

[4] Johan Fabry and Tom Mens. Language-independent detection of object-oriented design patterns. In *Proceedings of the European Smalltalk User Group's conference*, 2003. (Conditionally accepted).

[5] Adele Goldberg and Dave Robson. *Smalltalk-80: the language*. Addison-Wesley, 1983.

[6] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference of Aspect-Oriented Software Development*, 2003.

[7] IBM. Business rules for electronic commerce: Project at IBM T.J. Watson research, 1999. http://www.research.ibm.com/rules/.

[8] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. Rbcl: a reflective object-oriented concurrent language without a runtime kernel. In *IMSA'92 International Workshop on Reflection and Meta-Level Architectures*, 1992.

[9] Maxim Kizub. *Kiev language specification*, July 1998. http://www.forestro.com/kiev/kiev.html.

[10] Wilf R. LaLonde and Mark Van Gulik. Building a backtracking facility for Smalltalk without kernel support. In *Proceedings of the conference on Object-Oriented Languages, Systems and Applications*. ACM Press, 1988.

[11] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In *Proceedings of the 13th SEKE Conference*, 2001.

[12] Francois Pachet. On the embeddability of production rules in object-oriented systems. *Journal of Object-Oriented Programming*, 8(4), 1995.

[13] Patrick Steyaert. *Open Design of Object-Oriented Languages*. PhD thesis, Vrije Universiteit Brussel, 1994.

[14] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS-USA 1998*, 1998.

[15] Roel Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[16] Roel Wuyts and Kim Mens. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 1999*, 1999.

Unifying Tables, Objects and Documents

Erik Meijer[1], Wolfram Schulte[2], and Gavin Bierman[3]

[1] Microsoft Corporation, USA. emeijer@microsoft.com
[2] Microsoft Research, USA. schulte@microsoft.com
[3] Microsoft Research, UK. gmb@microsoft.com

Abstract. This paper proposes a number of type system and language extensions to natively support relational and hierarchical data within a statically typed object-oriented setting. In our approach SQL tables and XML documents become first class citizens that benefit from the full range of features available in a modern programming language like C$^\sharp$ or Java. This allows objects, tables and documents to be constructed, loaded, passed, transformed, updated, and queried in a unified and type-safe manner.

1 Introduction

The most important current open problem in programming language research is to increase programmer productivity, that is to make it easier and faster to write correct programs [29]. The integration of data access in mainstream programming languages is of particular importance—millions of programmers struggle with this every day. Data sources and sinks are typically XML documents and SQL tables but they are currently rather poorly supported in common object-oriented languages.

This paper addresses how to integrate tables and documents into modern object-oriented languages by providing a novel type system and corresponding language extensions.

1.1 The need for a unification

Distributed web-based applications are typically structured using a three-tier model that consists of a *middle tier* that contains the business logic that extracts relational data from a *data services tier* and processes it into hierarchical data that is displayed in the *user interface tier* (alternatively, in a B2B scenario, this hierarchical data might simply be transferred to another application). The middle tier is typically programmed in an object-oriented language such as Java or C$^\sharp$.

As a consequence, middle tier programs have to deal with relational data (SQL tables), object graphs, and hierarchical data (HTML, XML). Unfortunately these three different worlds are not very well integrated. As the following ADO.Net based example shows, access to a database usually involves sending a string representation of a SQL query over an explicit connection via a stateful API and then iterating over a weakly typed representation of the result set:

```
SqlConnection Conn = new SqlConnection(...);
SqlCommand Cmd = new SqlCommand
 ("SELECT Name, HP FROM Pokedex", Conn);
Conn.Open();
SqlDataReader Rdr = Cmd.ExecuteReader();
```

HTML or XML documents are then created by emitting document fragments in string form, without separating the model and presentation:

```
while (Rdr.Read()) {
 Response.Write("<tr><td>");
 Response.Write(Rdr.GetInt32(0));
 Response.Write("</td><td>");
 Response.Write(Rdr.GetString(1));
 Response.Write("</td></tr>");
}
```

Communication between the different tiers using untyped strings is obviously very fragile with lots of opportunities for silly errors and no possibility for static checking. In fact, representing queries as strings can be a security risk (the so-called ‘script code injection’ problem). Finally, due to the poor integration, performance suffers badly as well.

1.2 Previous attempts

It is not an easy task to gracefully unify the worlds of objects, documents and tables, so it should not come as a surprise that no main-stream programming language has yet emerged that realizes this vision.

Often language integration only deals with SQL *or* with XML but not with both [4, 14, 6, 9, 18]. Alternatively they start from a completely new language such as XQuery or XDuce [3, 12], which is a luxury that we cannot afford. Approaches based on language binding using some kind of pre-compiler such as XSD.exe, Castor, or JAXB do not achieve a real semantic integration. The impedance mismatch between the different type systems then leads to strange anomalies or unnatural mappings. Another popular route to integrate XML and SQL is by means of domain specific embedded languages [13] typically using a functional language such as Scheme or Haskell [26, 27, 22, 23, 19, 15, 10, 32, 33, 4] as the host. In our experience however, the embedded domain specific language approach does not scale very well, and it is particularly difficult to encode the domain specific type systems [30] and syntax into the host language.

1.3 The Xen solution

The examples above demonstrate that at a foundational level there is an impedance mismatch between the XML, SQL and object data-models. In our opinion the impedance mismatch is too big to attempt a complete integration. (The impedance mismatch between the object and XML data-models is treated in detail in a companion paper [25].)

Given these problems, our approach is to first take as our starting point the type system and object data-model of the middle tier programming language. This is the computational model that programmers are familiar with, and that is supported by the underlying execution engine.

We look at XML fidelity in terms of being able to serialize and deserialize as many possible documents that are expressible by some given XML schema language, and not at how closely we match one of the XML data models in our programming language once we have parsed an XML document. In other words, we consider XML 1.0 as simply syntax for serialized object instances of our enriched host language. For SQL fidelity we take the same approach: we require that SQL tables can be passed back and forth without having the need to introduce additional layers like ADO.NET.

Hence, rather than trying to blindly integrate the whole of the XML and SQL data-models, we enrich the type system of the object-oriented host language (in our case C$^\sharp$) with a small number of new type constructors such as streams, tuples, and unions. These have been carefully designed so that they integrate coherently with the existing type system.

On top of these type system extensions, we then add two new forms of expressions to the base language: generalized member access provides path expressions to traverse hierarchical data, and comprehension queries to join elements from different collections. Rather than limiting comprehension queries to tabular data or path expressions on hierarchical data, we allow both forms of expressions to be used on any collection, no matter whether the data is in-memory or remotely stored in a database. To seamlessly handle queries on both on remote data sources and local data sources we use similar deferred execution implementation techniques as in HaskellDB [19]. Depending on the data source, the result of a query is either a materialized collection or a SQL program that can be sent to the database.

The result is Xen, a superset of C$^\sharp$ that seamlessly blends the worlds of objects, tables and documents. The code fragment below shows how Xen is able to express the same functionality that we have seen previously.

```
tr* pokemon =
  select <tr><td>{Name}</td><td>{HP}</td></tr>
  from Pokedex;

Table t =
 <table><tr><th>Name</th><th>HP</th></tr>{pokemon} </table>;

Response.Write(t);
```

In Xen, strongly typed XML values are first-class citizens (i.e. the XML literal `<table>...</table>` has type static `Table`) and SQL-style `select` queries are built-in. Xen thus allows for static checking, and because the SQL and XML type systems are integrated into the language, the compiler can do a better job at generating efficient code that might run on the client but which also might be sent to the server.

Although Xen by design does not support the entirety of the XML stack and some of the more advanced features of SQL, we believe that our type system and language extensions are rich enough to support many potential scenarios. For example we have been able to program the complete set of XQuery Use Cases, and several XSL stylesheets, and we can even serialize the classic XML Hamlet document without running into any significant fidelity problems.

The next sections show how we have grown a modern object-oriented language (we take C$^\sharp$ as the host language, but the same approach will work with Java, Visual Basic, C++, etc.) to encompass the worlds of tables and documents by adding new types (§2) and expressions (§3).

2 The Xen type system

In this section we shall cover the extensions to the C$^\sharp$ type system—streams, tuples, discriminated unions, and content classes—and for each briefly consider the new query capabilities. In contrast to nominal types such as classes, structs, and interfaces, the new Xen types are mostly *structural* types, like arrays in Java or C$^\sharp$.

We will introduce our type system extensions by example. Formal details of the Xen type system can be found in a companion paper [24].

2.1 Streams

Streams represent ordered homogeneous collections of zero or more values. In Xen streams are most commonly generated by `yield return` blocks. Stream generators are like ordinary methods except that they may yield multiple values instead of returning a single time. The following method `From` generates a finite stream of integers $n, n+1, ..., m$:

```
static int* From(int n, int m){
  while(n<=m) yield return n++;
}
```

From the view of the host language, streams are typed refinements of C$^\sharp$'s *iterators*. Iterators encapsulate the logic for enumerating elements of collections.

Given a stream, we can iterate over its elements using C$^\sharp$'s existing `foreach` statement. For instance, the following loop prints the integers from n to m.

```
foreach(int i in From(n,m)) { Console.WriteLine(i); }
```

Streams and generators are not new concepts. They are supported by a wide range of languages in various forms [11, 20, 17, 21, 28]. Our approach is a little different in that:

- We classify streams into a hierarchy of streams of different length.
- We automatically flatten nested streams.
- We identify the value `null` with the empty stream.

To keep type-checking tractable, we restrict ourselves to the following stream types: $T*$ denotes possibly empty and unbounded streams, $T?$ denotes streams of at most one element, and $T!$ denotes streams with exactly one element. We will use $T?$ to represent optional values, where the non-existence is represented by the value `null` and analogously we use $T!$ to represent non-null values.

The different stream types form a natural subtype hierarchy, where subtyping corresponds to stream inclusion We write $S <: T$ to denote that type S is a subtype of type T and we write $S \cong T$ to denote that S is equivalent to T. Xen observes the axioms $T! <: T$, $T <: T?$ and $T? <: T*$. For instance $T? <: T*$ reflects the fact that a stream of at most one element is also a stream of at least zero elements. Non-stream types T into the subtype hierarchy by placing them between non-null values $T!$ and possibly null values $T?$. Thus allows for example to assign the value `3` to the type `int?`.

Like C$^\sharp$ arrays, streams are covariant. For unbounded streams the upcast of the elements is via an identity conversion. This restriction guarantees that upcasts of streams are always constant time, and that the object identity of the stream is maintained. Suppose `Button` is a subclass of `Control`, then this rule says that `Button*` is a subtype of a stream of controls `Control*`. If the conversion is not the identity, we have to explicitly copy the stream. For example, we can convert stream `xs` of type `int*` into a stream of type `object*` using the apply-to-all expression `xs.{ return (object)it; }`. Optional and non-null types are covariant with respect to arbitrary conversions on their element types.

In Xen streams are always flattened, there are no nested streams of streams. At the theoretical level this implies a number of type equivalences, for example $T*? \cong T*$ reflects the fact that at most one stream of zero or more elements flattens into a single stream of zero or more elements.

Flattening of stream types is essential to efficiently deal with recursively defined streams. Consider the following recursive variation of the function `From` that we defined previously:

```
int* From(int n, int m){
  if (n>m) {
    yield break;
  } else {
    yield return n++; yield return From(n,m);
  }
}
```

The recursive call `yield return From(n,m);` yields a stream forcing the type of `From` to be a nested stream. The non-recursive call `yield return n++;` yields a single integer thus forcing the return type of `From` to be a normal stream. As the type system treats the types `int*` and `int**` as equivalent this is type-correct.

Without flattening we would be forced to copy the stream produced by the recursive invocation, leading to a quadratic instead of a linear number of `yields`:

```
int* From(int n, int m){
  if (n >m) {
    yield break;
  } else {
    yield return n++;
```

```
      foreach(int it in From(n,m)) yield return it;
    }
  }
```

Flattening of stream types does *not* imply that the underlying stream is flattened via some coercion, every element in a stream is `yield`-ed at most once. Iterating over a stream effectively perform a depth-first traversal over the n-ary tree produced by the stream generators.

Non-nullness. The type $T!$ denotes streams with exactly one element, and since we identify `null` with the empty stream, this implies that values of type $T!$ can never be `null`.

Being able to express that a value cannot be `null` via the type system allows *static* checking for `null` pointers (see [7, 8] for more examples). This turns many (potentially unhandled) dynamic errors into compile-time errors.

One of the several methods in the .NET base class library that throws an `ArgumentNullException` when its argument is `null` is the function `IPAddress.Parse`. Consequently, the implementation of `IPAddress.Parse` needs an explicit `null` check:

```
  public static IPAddress Parse(string ipString) {
    if (ipString == null)
      throw new ArgumentNullException("ipString");
    ...
  }
```

Dually, clients of `IPAddress.Parse` must be prepared to catch an `ArgumentNullException`. Nothing of this is apparent in the type of the `Parse` method in C$^\sharp$. In Java the signature of `Parse` would at least show that it possibly throws an exception.

It would be much cleaner if the *type* of `IPAddress.Parse` indicated that it expects its `string` argument to be non-`null`:

```
  public static IPAddress Parse(string! a);
```

Now, the type-checker statically rejects any attempt to pass a string that might be `null` to `IPAddress.Parse`.

2.2 Anonymous structs

Tuples, or *anonymous structs* as we call them, encapsulate heterogeneous ordered collections values of fixed length. Members of anonymous structs can optionally be labelled, and labels can be duplicated, even at different types. Members of anonymous structs can be accessed by label or by position. Anonymous structs are value types, and have no object identity.

The function `DivMod` returns the quotient and remainder of its arguments as a tuple that contains two named integer fields `struct{int Div, Mod;}`:

```
struct{int Div, Mod;} DivMod(int x, int y) {
  return new(Div = x/y, Mod = x%y);
}
```

The members of an anonymous struct may be unlabelled, for example, we can create a tuple consisting of a labelled `Button` and an unlabelled `TextBox` as follows:

```
struct{Button enter; TextBox;} x =
  new(enter=new Button(), new TextBox());
```

An unlabelled member of a *nominal* type is a shorthand for the same member implicitly labelled with its type.

As mentioned earlier, members of tuples can be accessed either by position, or by label. For example:

```
int m = new(47,11)[0];
Button b = x.enter;
```

As for streams, tuples are covariant provided that the upcast-conversion that would be applied is the identity. Subtyping is lifted over field declarations as expected. This means that we can assign `new(enter=new Button(), new TextBox())` to a variable of type `struct{Control enter; Textbox;}`.

2.3 Streams+anonymous structs = tables

Relational data is stored in tables, which are sets of rows. Sets can be represented by streams, and rows by anonymous structs, thus streams and anonymous structs together can be used to model relational data.

The table below contains some basic facts about Pokemon characters such as their name, their strength, their kind, and the Pokemon from which they evolved (see `http://www.pokemon.com/pokedex/` for more details about these interesting creatures).

Name	HP	Kind	Evolved
Meowth	50	Normal	
Rapidash	70	Fire	Ponyta
Charmelon	80	Fire	Charmander
Zubat	40	Plant	
Poliwag	40	Water	
Weepinbell	70	Plant	Bellsprout
Ponyta	40	Fire	

This table can be modelled by the variable `Pokedex` below:

```
enum Kind {Water, Fire, Plant, Normal, Rock}

struct{string Name; int HP; Kind Kind; string? Evolved;
}* Pokedex;
```

The fact that basic Pokemon are not evolutions of other Pokemon shows up in that the `Evolved` column has type `string?`.

2.4 Discriminated union

A value of a *discriminated union* holds (at different times) any of the values of its members. Like anonymous structs, the members of discriminated unions can be labelled or unlabelled.

Discriminated unions often appear in content classes (see §2.5 below). The type `Address` uses a discriminated union to allow either a member `Street` of type `string` or a member `POBox` of type `int`:

```
class Address {
  struct{
    choice{ string Street; int POBox; };
    string City; string? State; int Zip;
    string Country;
  };
}
```

The second situation in which discriminated unions are used is in the result types of generalized member access (see §3.2). For example, when `p` has type `Pokemon`, the wildcard expression `p.*` selects all members of `p` which returns a stream containing all the members of a Pokemon and has type

```
choice{string; int; Kind; string?}*
```

Using the subtype rules for `choice` and streams this is equivalent to `choice{string?; int; Kind;}*`.

Unlike unions in C/C++ and variant records in Pascal where users have to keep track of which type is present, values of an discriminated unions in Xen are implicitly tagged with the static type of the chosen alternative, much like unions in Algol68. In other words, discriminated unions in Xen are essentially a pair of a value and its static type. The type component can be tested with the conformity test e `was` T. The expression e `was` T is true for *exactly one* T in the union. This invariant is maintained by the type system. You can get the value component of a discriminated union value by downcasting.

Labelled members of discriminated unions are just nested singleton anonymous structs, for example `choice{int Fahrenheit; int Celsius;}` is a shorthand for the more verbose `choice{struct{int Fahrenheit;}; struct{int Celsius;};}`. Discriminated unions are idempotent (duplicates are removed), associative and commutative (nesting and order are ignored).

Values of non-discriminated unions can be injected into a discriminated union. This rule allows us to conveniently inject values into a discriminated union as in the example below:

```
choice{int Fahrenheit; int Celsius;} = new(Fahrenheit=47);
```

Finally, streams distribute over nested discriminated unions, Again this is essential for recursively defined streams as in the following example which returns a stream of integers terminated by `true`:

```
choice{int; bool;}* f(int n) {
  if(n==0){
    yield return true;
  } else {
    yield return n;
    yield return f(--n);
  }
}
```

2.5 Content classes

Now that we have introduced streams, anonymous structs, and discriminated unions, our type system is rich enough to model a large part of the XSD schema language; our aim is to cover as much of the essence of XSD [31] as possible whilst avoiding most of its complexity.

The correspondence between XSD particles such as `<sequence>` and `<choice>` with local element declarations and the type constructors `struct` and `choice` with (labelled) fields should be intuitively clear. Likewise, the relationship of XSD particles with occurrence constraints to streams is unmistakable. For $T*$ the attribute pair `(minOccurs, maxOccurs)` is `(0, unbounded)`, for $T?$ it is `(0, 1)`, and for $T!$ it is `(1,1)`.

The content class `Address` that we defined in §2.4 corresponds to the XSD schema `Address` below:

```
<element name="Address"><complexType>
  <sequence>
   <choice>
    <element name="Street" type="string">
    <element name="POBox" type="integer">
   </choice>
   <element name="City" type="string">
   <element name="State" type="string" minOccurs="0"/>
   <element name="Zip" type="integer"/>
   <element name="Country" type="string"/>
  </sequence>
</complexType></element>
```

A Xen content class is simply a normal C$^\sharp$ class with a single unlabelled member and zero or more methods. As a consequence, the content can only ever be accessed via its individually named children, which allows the compiler to choose the most efficient data layout.

The next example schema defines two top level elements `Author` and `Book` where `Book` elements can have zero or more `Author` members:

```
<element name="Author"><complexType>
  <sequence>
    <element name="Name" type="string"/>
  </sequence>
</complexType></element>

<element name="Book"><complexType>
  <sequence>
   <element name="Title" type="string"/>
   <element ref="Author" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</complexType></element>
```

In this case, the local element reference is modelled by an unlabelled field and the two elements are mapped onto the following two type declarations:

```
  class Author { string Name; }
  class Book { struct{ string Title; Author*; } }
```

All groups such as the one used in the following schema for the complex type `Name`

```
  <element name="Name"><complexType>
    <all>
     <element name="First" type="string"/>
     <element name="Last" type="string"/>
    </all>
  </complexType></element>
```

are mapped to ordinary fields of the containing type:

```
 class Name { string First; string Last; }
```

As these examples show, both top-level element declarations and named complex type declarations are mapped to top-level types. This allows us to unify derivation of complex types and substitution groups of elements using standard inheritance. Further details of the relationship between the XML and Xen data-models can be found in a companion paper [25].

3 Xen expressions

In the previous sections we have concentrated on the Xen type system. In this section we will consider new Xen expression forms to construct, transform, query and combine Xen values.

3.1 XML constructors

Xen internalizes XML serialized objects into the language, conveniently allowing programmers to use XML fragments as object literals. For instance, we can create a new instance of an `Address` object using the following XML object literal:

```
Address a = <Address>
              <Street>One Microsoft Way</Street>
              <City>Redmond</City>
            </Address>;
```

The Xen compiler contains a validating XML parser that analyzes the XML literal and “deserializes” it at compile time into code that will construct the correct `Address` instance. This allows Xen programmers to treat XML fragments as first-class values in their code.

XML literals can also have placeholders to describe *dynamic* content (anti-quoting). We use the XQuery convention whereby an arbitrary expression or statement block can be embedded inside an element by escaping it with curly braces:

```
Author NewAuthor(string name) {
  return <Author>{name.ToUpper()}</Author>;
}
```

Embedded expressions must return or yield values of the required type (in this case `string`). Validation of XML literals with placeholders is non-trivial and is the subject of a forthcoming paper.

Note that XML literals are treated by Xen as just object constructors, there is nothing special about content classes. In fact, we can write XML literals to construct values of any type, for example, the assignment

```
Button b = <Button>
             <Text>Click Me</Text>
           </Button>;
```

creates an instance of the standard `Button` class and sets its `Text` field to the string `"Click Me"`.

3.2 Stream generators, iterators and lifting

To make the creation of streams as concise as possible, we allow *anonymous method bodies* as expressions. In the example below we assign the (conceptually) infinite stream of positive integers to the variable `nats`:

```
// 0, 1, 2, ...
int* nats = { int i=0; while(true) yield return i++; };
```

Our stream constructors (`*`, `?`, `!`) are functors, and hence we implicitly *lift* operations on the element type of a stream (such as member or property access and method calls) over the stream itself. For instance, to convert each individual string in a stream `ss` of strings to uppercase, we can simply write `ss.ToUpper()`.

We do not restrict this lifting to member access. Xen generalizes this with an *apply-to-all* block. We can write the previous example as `ss.{ return it.ToUpper(); }`. The implicit argument `it` refers successively to each element of the stream `Ss`.

In fact, the apply-to-all block itself can yield a stream, in which case the resulting nested stream is flattened in the appropriate way. For example (where nats is a stream of integers):

```
// 1, 2,2, 3,3,3, 4,4,4,4, ...
int* rs = nats.{ for(i=0; i<it; i++) yield return it; };
```

If an apply-to-all block returns void, no new stream is constructed and the block is eagerly applied to all elements of the stream. For example to print all the elements of a stream we can just write:

```
nats.{ Console.WriteLine(it); };
```

Apply-to-all blocks can be stateful, so we can use them to do reductions (in the functional community called *folds*). For example, we can sum all integers in an integer stream xs as follows:

```
int sum(int* xs){
  int s = 0; xs.{ s += it; }; return s;
}
```

We need to be careful when lifting over non-null types, since the fact that the receiver object is not null does not imply that its members are not null either:

```
Button! b = <Button/>;
Control p = b.Parent; // Parent might be null
```

Hence the return type of lifting over a non-null type is not guaranteed to return a non-null type.

Optional types provide a standard implementation of the null design pattern; when a receiver of type $T?$ is null, accessing any of its members returns null:

```
string? t = null;
int? n = t.Length; // n = null
```

In Objective-C [16] this is the standard behaviour for any object that can be null.

Member access is not only lifted over streams, but over all structural types. For example the expression xs.x will return the stream true, 1, 2 of type choice{bool; int;}+ when xs is defined as:

```
struct{ bool x; struct{int x;}*; } xs =
  new( x=true
     , {yield return new(x=1); yield return new(x=2);}
     );
```

Lifting over discriminated unions introduces a possibility of nullness for members that are not in all of the alternatives. Suppose x has type choice{ int; string; }. Since only string has a Length member, the type of x.Length is int? which reflects the fact that in case the dynamic type of x is int, the result of x.Length will be null. Since int and string both have a member GetType(), the return type of x.GetType() is Type:

```
choice{ int; string; } x = 4711;
int? n = x.Length;    // null
Type t = x.GetType(); // System.Int32
```

In case the alternatives of a union have a member of different type in common, the result type is the union of the types of the respective members.

Binary and unary operators are lifted element-wise over streams. For example we can add two optional integers `x+y` to get another optional integer. If either `x` or `y` is `null` the result of adding them is `null` as well. Lifting of optional types implements SQL's three-value logic.

Often we want to *filter* a stream according to some predicate on the elements of the stream. For example, to construct a stream with only odd numbers, we filter out all even numbers from the stream `nats` of natural numbers using the filter expression `nats[it%2==1]`. For each element in the stream to be filtered, the predicate is evaluated with that element bound to `it`. Only if the predicate is true the element becomes part of the new stream.

```
int* odds1 = nats[it%2 == 1];
```

In fact, filters can be encoded using an apply-to-all block:

```
int* odds2 = nats.{if(it%2 == 1) yield return it;};
```

3.3 Further generalized member access

As we have seen, Xen elegantly generalizes familiar C$^\sharp$ member access resulting in compact and clear code. However we should like to provide more flexible forms of member access: Xen provides *wildcard*, *transitive* and *type-based* access. These forms are similar to the concepts of nametest, abbreviated relative location paths and name filters in XPath [1], but have been adapted to work uniformly on object graphs.

Wildcards provide access to all members of a type without needing to specify the labels. For example, suppose that we want to have all fields of an `Address`:

```
choice{string; int;}* addressfields = Microsoft.*;
```

The wildcard expression returns the content of all accessible fields and properties of the variable `Microsoft` in their declaration order. In this case `"One Microsoft Way"`, `"Redmond"`, `98052`, `"USA"`.

Transitive member access, written as `e...m`, returns all accessible members m that are transitively reachable from `e` in depth-first order. The following declaration of `authors` (lazily) returns a stream containing all `Author`s of all `Book`s in the source stream `books`:

```
Book F = <Book>
          <Title>Faust</Title>
          <Author>Goethe</Author>
         </Book>;
Book K = <Book>
```

```
            <Title>De Klompeniers</Title>
            <Author>Jac. Broersen</Author>
          </Book>;

Book* books = { yield F; yield K; };
string* authors = books...Author;
```

Transitive member access abstracts from the concrete representation of a tree; as long as the mentioned member is reachable and accessible, its value is returned.

Looking for just a field name might not be sufficient, especially for transitive queries where there might be several reachable members with the same name, but of different type. In that case we allow an additional type-test to restrict the matching members. A type-test on T selects only those members whose static type is a subtype of T. For instance, if we are only interested in Microsoft's `POBox` number, and `Zip` code, we can write the transitive query `Microsoft...int::*`.

3.4 Comprehensions

The previous sections presented our solutions to querying documents. However for accessing relational data, which we model as streams of anonymous structs, simple SQL queries are more natural and flexible. Here we only consider the integration of the SQL `select-from-where` clause, and defer the discussion of more advanced features such as data manipulation and transactions to a future paper.

The fundamental operations of relational algebra are *selection*, *projection*, *union*, *difference* and *join*. Here are two simple SQL-style comprehension queries:

```
Pokemon* ps1 =
  select * from Pokedex where Kind == Normal;
struct{string Name; Kind Kind;}* ps2 =
    select Name, Kind from Pokedex;
```

In practice, the result types of SQL queries can be quite involved and hence it becomes painful for programmers to explicitly specify types. Since the compiler already knows the types of sub-expressions, the result types of queries can be inferred automatically. Providing type declarations for method local variables is not necessary, and we can simply write:

```
ps2 = select Name, Kind from Pokedex;
```

without having to declare the type of `ps2`.

Union and difference present no difficulty in our framework. They can easily be handled with existing operations on streams. Union concatenates two streams into a single stream. Difference takes two streams, and returns a new stream that contains all values that appear in the first but not in the second stream.

The real power of comprehensions comes from join. Join takes two input streams and creates a third stream whose values are composed by combining members from the two input streams. For example, here is an expression that selects pairs of Pokemons that have evolved from each other:

```
select p.Name, q.Name
from p in Pokedex, q in Pokedex
where p.Evolved == q
```

Again, we should like to emphasize the elegant integration of data in Xen. The select expression works on arbitrary streams, whether in memory or on the hard disk; streams simply virtualize data access. Strong typing makes data access secure. But there is no excessive syntactic burden for the programmer as the result types of queries are inferred.

4 Conclusion

The language extensions proposed in this paper support both the SQL [2] and the XML schema type system [31] to a large degree, but we have not dealt with all of the SQL features such as (unique) keys, and the more esoteric XSD features such as redefine. Similarly, we capture much of the expressive power of XPath [1], XQuery [3] and XSLT [5], but we do not support the full set of XPath axis. We are able to deal smoothly with namespaces, attributes, blocking, and facets however. Currently we are investigating whether and which additional features need to be added to our language.

In summary, we have shown that it is possible to have both SQL tables and XML documents as first-class citizens in an object-oriented language. Only a bridge between the type worlds is needed. Building the bridge is mainly an engineering task. But once it is available, it offers the best of three worlds.

Acknowledgments

We should like to acknowledge the support, encouragement, and feedback from Mike Barnett, Nick Benton, Don Box, Luca Cardelli, Bill Gates, Steve Lucco, Chris Lucas, Todd Proebstring, Dave Reed, Clemens Szyperksi, and Rostislav Yavorskiy and the hard work of the WebData languages team consisting of William Adams, Joyce Chen, Kirill Gavrylyuk, David Hicks, Steve Lindeman, Chris Lovett, Frank Mantek, Wolfgang Manousek, Neetu Rajpal, Herman Venter, and Matt Warren. This paper was written whilst Bierman was in the University of Cambridge Computer Laboratory and supported by EU AppSem II.

Bibliography

[1] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML path language 2.0. `http://www.w3.org/TR/xpath20/`.

[2] G.M. Bierman and A. Trigoni. Towards a formal type system for ODMG OQL. Technical Report 497, University of Cambridge Computer Laboratory, 2000.

[3] S. Boag, D. Chamberlin, M.F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery/`.

[4] A.S. Christensen, A. Muller, and M.I. Schwartzbach. Static analysis for dynamic XML. In *Proceedings of PlanX*, 2002.

[5] J.J. Clark. XSL Transformations 1.0. `http://www.w3.org/TR/xslt`.

[6] R. Connor, D. Lievens, and F. Simeoni. Projector: a partially typed language for querying XML. In *Proceedings of PlanX*, 2002.

[7] M. Fahndrich and R.M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of OOPSLA*, 2003.

[8] C. Flanagan, R. Leino, M. Lillibridge, C. Nellson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI*, 2002.

[9] V. Gapeyev and B.C. Pierce. Regular object types. In *Proceedings of ECOOP*, 2003.

[10] P. Graunke, S. Krishnamurthi, S.V.D. Hoeven, and M. Felleisen. Programming the web with high-level programming languages. In *Proceedings of ASE*, 2001.

[11] R. Griswold and M. Griswold. *The Icon programming language*. Prentice Hall, 1990.

[12] H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language (preliminary report). In *Proceedings of WebDB*, number 1997 in LNCS, pages 226–244, 2000.

[13] P. Hudak. Building domain specific embedded languages. *ACM computing surveys*, 28(4), 1996.

[14] M. Kempa and V. Linnemann. On XML objects. In *Proceedings of PlanX*, 2002.

[15] O. Kiselyov and S. Krishnamurthi. SXSLT: A manipulation language for XML. In *Proceedings of PADL*, 2003.

[16] S. Kochan. *Programming in Objective C*. Sams, 2003.

[17] A. Krall and J. Vitek. On extending Java. In *Proceedings of JMLC*, 1997.

[18] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating structured and semistructured data. In *Proceedings of DBPL*, 2000.

[19] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of USENIX Conference on Domain-specific languages*, 1999.

[20] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C.Schaffert, R. Scheiffer, and A. Snyder. *CLU reference manual*. Springer Verlag, 1981.

[21] B. Liskov, M.Day, M. Herlihy, P. Johnson, and G. Leavens. ARGUS reference manual. Technical report, MIT, 1987.

[22] E. Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 10(1), 2000.

[23] E. Meijer, D. Leijen, and J. Hook. Client-side web scripting with HaskellScript. In *Proceedings of PADL*, 2002.

[24] E. Meijer, W. Schulte, and G.M. Bierman. The essence of Xen. Submitted for publication, 2003.

[25] E. Meijer, W. Schulte, and G.M. Bierman. Programming with circles, triangles and rectangles. In *Proceedings of XML 2003*, 2003.

[26] E. Meijer and M. Shields. XMλ: a functional language for constructing and manipulating XML documents. Unpublished paper, 1999.

[27] E. Meijer and D. van Velzen. Haskell server pages. In *Proceedings of Haskell workshop*, 2000.

[28] S. Murer, S. Omohundro, D. Stoutamire, and C.Szyperski. Iteration abstraction in Sather. *ACM ToPLAS*, 18(1):1–15, 1996.

[29] T.A. Proebsting. Disruptive programming language technologies. Unpublished note, 2002.

[30] M. Shields and E. Meijer. Type-indexed rows. In *Proceedings of POPL*, 2001.

[31] J. Simeon and P. Wadler. The essence of XML. In *Proceedings of POPL*, 2003.

[32] P. Thiemann. WASH/CGI: Server side web scripting with sessions and typed compositional forms. In *Proceedings of PADL*, 2002.

[33] N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two little languages. In *Proceedings of Workshop on Scheme and functional programming*, 2002.

Syntax sugar for FC++: lambda, infix, monads, and more

Brian McNamara and Yannis Smaragdakis

College of Computing
Georgia Institute of Technology
`http://www.cc.gatech.edu/∼yannis/fc++/`
`lorgon,yannis@cc.gatech.edu`

Abstract. We discuss the FC++ library, a library for functional programming in C++. We give an overview of the library's features, but focus on recent additions to the library. These additions include the design of our "lambda" sublanguage, which we compare to other lambda libraries for C++. Our lambda sublanguage contains special syntax for programming with monads, which we also discuss in detail. Other recent additions which we discuss are "infix function syntax" and "full functoids".

1 Introduction

FC++[7, 8] is a library for functional programming in C++. We have recently added a number of new features to the FC++ library, most notably an expression template library for creating a *lambda* sublanguage. The lambda sublanguage contains special syntax for programming with *monads* in the style of Haskell. We focus our discussion on the design of this portion of the library (Section 5 and Section 6), but begin with a run-down of the features of FC++ (Section 2 and Section 3) as well as some important implementation details (Section 4).

2 Overview

In FC++, programmers define and use *functoids*. Functoids are the FC++ representation of functions; we will discuss them in more detail in Section 4. The latest version (v1.5) of the FC++ library supports a number of useful features, including

- higher order, polymorphic functoids ("direct" functoids)
- lazy lists
- a large library of functoids, combinators, and monads (most of which duplicate a good portion of the Haskell Standard Prelude[2])
- currying
- infix functoid syntax
- dynamically-bound functoids ("indirect" functoids)
- a small library of effect combinators
- interfaces to C++ Standard Library data structures and algorithms via iterators

– ways to transform methods of classes and normal C++ functions into functoids
– reference-counted “smart” pointers for memory management (used internally by, e.g., our lazy list data structure)

We’ll briefly discuss each of these features in the next section. Later on we will discuss

– special syntax to mimic functional language constructs, including *lambda*, *let*, and *letrec*, as well as *do*-notation and *comprehensions* for arbitrary monads

in detail.

The FC++ library is about 9000 lines of C++ code, and is written with strict conformance to the C++ standard[4], which makes it portable to all of the major brands of compilers.

3 Short Examples of various features

FC++ functoids can be simultaneously higher order (able to take functoids as arguments and return them as results) and polymorphic (template functions which work on a variety of data types). For example, consider the library function `compose()`, which takes two functoids and returns the composition:

```
// compose( f, g )(args)  ==  f( g(args) )
```

We could define a polymorphic functoid `addSelf()`, which adds an argument to itself:

```
// addSelf( x )  ==  x + x
```

We could then compose `addSelf` with itself, and the result would still be a polymorphic functoid:

```
int x = 3;
std::string s = "foo";
compose( addSelf, addSelf )( x )   // yields 12
compose( addSelf, addSelf )( s )   // yields "foofoofoofoo"
```

Section 4 describes the infrastructure of these “direct functoids”, which enables this feat to be implemented.

FC++ defines a lazy list data structure called `List`. `List`s are lazy in that they need not compute their elements until they are demanded. For example, the functoid `enumFrom()` takes an integer and returns the infinite list of integers starting with that number:

```
enumFrom( 1 )     // yields infinite list [1,2,3,...]
```

A number of functoids manipulate such lists; for instance `map()` applies a functoid to each element of a list:

```
map(addSelf, enumFrom( 1 ))  // yields infinite list [2,4 6,...]
```

The FC++ library defines a wealth of useful functoids and data types. There are named functoids for most C++ operators, like

```
plus(3,4)       // 3+4      also minus, multiplies, etc.
```

There are many functoids which work on `List`s, including `map`. Most of the `List` functions are identical those defined in Haskell[2]. Additionally, a number of basic functions (like the identity function, `id`), combinators (like `flip`: `flip(f)(x,y)==f(y,x)`), and data types (like `List` and `Maybe`; `Maybe` will be discussed in Section 6) are designed to mimic exactly their Haskell counterparts. We also implement functoids for such C++ constructs as constructor calls and `new` calls:

```
construct3<T>()(x,y,z)   // yields T(x,y,z)
new2<T>()(x,y)           // yields new T(x,y)
```

and many more (some of which are described below).

Functoids are curryable. That is, we can call a functoid with some subset of its arguments, returning a new functoid which expects the rest of the arguments. Currying of leading arguments can be done implicitly, as in

```
minus(3)      // yields a new function "f(x)=3-x"
```

Any argument can be curried explicitly using the placeholder variable `_` (defined by FC++):

```
  minus(3,_)    // yields a new function "f(x)=3-x"
  minus(_,3)    // yields a new function "f(x)=x-3"
```

We can even curry all *N* of a function's arguments with a call to `curryN()`, returning a *thunk* (a zero-argument functoid):

```
  curry2( minus, 3, 2 )   // yields a new thunk "f()=3-2"
```

FC++ functoids can be called using a special infix syntax (implemented by overloading `operator^`):

```
x ^f^ y        // Same as f(x,y).  Example: 3 ^plus^ 2
```

This syntax was also inspired by Haskell; some function names (like `plus`) are more readable as infix than as prefix.

FC++ defines *indirect functoids*, which are function variables which can be bound to any function with the same (monomorphic) signature. Indirect functoids are implemented via the `FunN` classes, which take *N* template arguments describing the argument types, as well as a template argument describing the result type. For example:

```
// Note: plus is polymorphic, the next line selects just
//       "int" version
Fun2<int,int,int> f = plus;
f(3,2);          // yields 5
f = minus;
f(3,2);          // yields 1
```

Indirect functoids are particularly useful in the implementation of callback libraries and some design patterns[11].

The FC++ library defines a number of effect combinators. An effect combinator combines an effect (represented as a thunk) with another functoid. Here are some example effect combinators:

```
// before(thunk,f)(args) == { thunk(); return f(args); }
// after(g,thunk)(args)  == { R r = g(args); thunk(); return r; }
```

An example: suppose you've defined a functoid `writeLog()` which takes a string and writes it to a log file. Then

```
 before( curry1( writeLog, "About to call foo()" ), foo )
```

results in a new functoid with the same behavior as `foo()`, only it writes a message to the log file before calling `foo()`.

FC++ interfaces with normal C++ code and the STL. The `List` class implements the iterator interface, so that lists can work with STL algorithms and other STL data structures can be converted into `List`s. The functoid `ptr_to_fun()` transforms normal C++ function pointers into functoids, and turns method pointers into functions which take a pointer to the receiver object as an extra first object. Here are some examples, which use currying to demonstrate that the result of `ptr_to_fun` is a functoid:

```
 ptr_to_fun( &someFunc )(x)(y)         // someFunc(x,y)
 ptr_to_fun( &Foo::meth )(aFooPtr)(x)  // aFooPtr->meth(x)
```

FC++ comes with its own reference-counted smart pointers: `Ref` and `IRef`. `Ref<T>` works just like a `T*`, only with reference counting. `IRef<T>` implements intrusive reference counting; an efficient form of reference counting which requires supportive help from the type being used (here, `T`). Internally, the library uses `IRef`s in the implementation of `List`s and indirect functoids.

4 Where is the magic?

In the previous section we saw how functoids can be used. Nevertheless, we have not shown you how the polymorphic functoids inside FC++ are implemented or how to define your own polymorphic functoids. In this section we show how functoids are defined, and how they gain the special functionality FC++ supports (like currying and infix syntax).

4.1 Defining polymorphic functoids

To create your own polymorphic functoid, you need to create a class with two main elements: a template `operator()` and a member structure template named `Sig`. To make things concrete, consider the definition of `map` (or rather, the class `Map`, of which `map` is a unique instance) shown in Figure 1. This definition uses the helper template `FunType`, which is a specialized template for different numbers of arguments. For two arguments, `FunType` is essentially:

```
template <class A1, class A2, class R> struct FunType {
  typedef R ResultType; typedef A1 Arg1Type; typedef A2 Arg2Type;  };
```

We can now analyze the implementation of `Map`. The `operator()` will allow instances of this class to be used with regular function call syntax. What is special in this case is that the operator is a template, which means that it can be used with

```
struct Map {
   template <class F, class L>
   struct Sig : public FunType<F,L,List<typename F::template
      Sig<typename L::ElementType>::ResultType> > {};

   template <class F, class T>
   typename Sig<F, List<T> >::ResultType
   operator()( const F& f, const List<T>& l ) const {
      if( null(l) )
         return NIL;
      else
         return cons( f(head(l)), curry2(Map(), f, tail(l)) );
   }
} map;
```

Fig. 1. Defining `map` in FC++

arguments of multiple types. When an instance of `Map` is used with arguments `f` and `l`, unification will be attempted between the types of `f` and `l`, and the declared types of the parameters (`const F&`, and `const List<T>&`). The unification will yield the values of the type parameters `F` and `T` of the template. This will determine the return type of the functoid.

Now, let's examine the `Sig` member class of the `Map` class. By FC++ convention, the `Sig` member should be a template over the argument types of the function you want to express (in this case the function type `F` and the list type `L`). The `Sig` member template is used to answer the question "what type will your function return if I pass it these argument types?" The answer in the `Map` code is:

```
   List< F::Sig<L::ElementType>::ResultType >
```

(we have elided the `typename` and `template` keywords for readability). This means: "map returns a `List` of what `F` would return if passed an element like the ones in list `L`".

In Haskell, one would express the type signature of `map` as:

```
   map :: (a -> b) -> [a] -> [b]
```

The `Sig` members of FC++ functoids essentially encode the same information, but in a computational form: `Sig`s are type-computing compile-time functions that are called by the C++ unification mechanism for function templates and implement the FC++ type system. This type system is completely independent from the native C++ type system—`map`'s type as far as C++ is concerned is just `class Map`. Other FC++ functoids, however, can read the FC++ type information from the `Sig` member of `Map` and use it in their own type computations. The `map` functoid itself uses that information from whatever functoid happens to be passed as its first argument (see the `F::Sig<L::ElementType>::ResultType` expression, above).

4.2 Using the FullN wrappers to gain functionality

The definition of map in the previous subsection creates what we call a “basic direct functoid” in FC++. However, a number of features of functoids (such as currying and infix syntax, which we saw in Section 3, and lambda-awareness, which will shall describe in Section 5) only work on so-called “full functoids”.

Transforming a normal functoid into a full functoid is easy. For example, to define map as a full functoid, we change the definition from Figure 1 from

```
struct Map { /* ... */ } map;
```

to

```
struct XMap { /* ... */ };
typedef Full2<XMap> Map;
Map map;
```

That is, FullN<F> is the type of the full functoid created out of the basic *N*-argument functoid F. The FullN template classes serve as a wrapper around basic functoids. They add all of the FC++ features we are accustomed to (such as currying and infix syntax) to the basic functoid.

Full functoids are a new feature of the FC++ library. Legacy code can promote its basic functoids into full functoids either by making the minor modification to the definition described above, or within an expression by calling the functoid makeFullN(), which takes an *N*-argument basic functoid as an argument and returns the corresponding full functoid as a result.

5 Lambda

Lambda is no stranger to C++. There are a number of existing C++ libraries which enable clients to create new, anonymous functions on-the-fly. Some such libraries, like the C++ STL[12] and its “binders”, or previous versions of FC++, allow the creation of new functoids on-the-fly only either by binding some subset of a functions arguments to values (currying) or by using combinators (like compose). Other libraries, like the Boost Lambda Library[5] and FACT![13] enable the creation of arbitrary lambdas by using expression templates.

5.1 Motivation

We were motivated to implement lambda by our interest in programming with monads. Experience with previous versions of FC++ made it clear that arbitrary lambdas are a practical necessity if one wants to program with monads. Older versions of FC++ had a number of useful combinators which made it possible to express most arbitrary functions, but lambda makes it practical by making it readable. For example, while implementing a monad, in the middle of an expression you might discover that you need a function with this meaning:

```
lambda(x) { f(g(x),h(x)) }
```

It is possible to implement this function using combinators (without lambda), but the resulting code is practically unreadable:

```
duplicate(compose(flip(compose)(h),compose(f,g)))
```

Alternatively, you can define the new functoid at the top level, give it a name, and then call it:

```
struct XFoo {
   template <class X> struct Sig : public FunType<X,
      typename RT<F<typename RT<G,X>::ResultType,
      typename RT<H,X>::ResultType>::ResultType> {};
   template <class X>
   typename Sig<X>::ResultType operator()( const X& x ) const {
      return f(g(x),h(x));
   }
};
typedef Full1<XFoo> Foo;
Foo foo;
// later use "foo"
```

but clearly this is way too much work, especially when the function in question is a one-time-use ("throwaway") function. Lambda is the only reasonable solution when you need to define short, readable, arbitrary functions on-the-fly.

5.2 Problematic issues with expression-template lambda libraries

Despite the advantages to lambda, we have always maintained a degree of wariness when it comes to C++ lambda libraries (or any expression template library), owing to the intrinsic limitations and caveats of using expression templates in C++. The worrisome issues with expression template libraries in general (or lambda libraries in particular) fall into four major categories:

– **Accidental/early evaluation.** The biggest problem with expression template lambda libraries comes from accidental evaluation of C++ expressions. Consider a short example using the Boost Lambda Library:

  ```
  int a[] = { 5, 3, 8, 4 };
  for_each( a, a+4, cout << _1 << "\n" );
  ```

  The third argument to `for_each()` creates an anonymous function to print each element of the array (one element per line). The output is what we would expect:

  ```
  5
  3
  8
  4
  ```

  If we want to add some leading text to each line of output, it is tempting to change the code like this:

  ```
  int a[] = { 5, 3, 8, 4 };
  for_each( a, a+4, cout << "Value: " << _1 << "\n" );
  ```

But (surprise!), the new program prints the added text only once (rather than once per line):

```
Value: 5
3
8
4
```

This is because "`cout << "Value: "`" is a normal C++ expression that the C++ compiler evaluates immediately. Only expressions involving placeholder variables (like `_1`)[1] get "delayed" from evaluation by the expression templates. These accidents are easy to make, and hard to see at a glance.

- **Capture semantics (lambda-specific).** Since C++ is an effect-ful language, it matters whether free variables captured by lambda are captured by-value or by-reference. The library must choose one way or the other, or provide a mechanism by which users can choose explicitly.
- **Compiler error messages.** C++ compilers are notoriously verbose when it comes to reporting errors in template libraries. Things are even worse with expression template libraries, both because there tend to be more levels of depth of template instantiations, and because the expression templates typically expose clients to some new/unfamiliar syntax, which makes it more likely for clients to make accidental errors. Indecipherable error messages may make an otherwise useful library be too annoying for clients to use.
- **Performance.** Expression template libraries sometimes take orders of magnitude longer to compile than comparably-sized C++ programs without expression templates. Also, the generated binary executables are often much larger for programs with expression templates.

For the most part, these problems are intrinsic to all expression template libraries in C++. As a result, when we set out to design a lambda library for FC++, we kept in mind these issues, and tried to design so as to minimize their impact.

5.3 Designing for the issues

Here are the design decisions we have made to try to minimize the issues described in the previous subsection.

- **Accidental/early evaluation.** Since the problem itself is intrinsic to the domain, the only way to "attack" this issue is prevention. That is, we cannot prevent users from making mistakes, but we can try to design our lambda to make these mistakes less common and/or more immediately apparent. To this end, we have designed the lambda syntax to be minimalist and visually distinct:

[1] Additionally, one can use other special constructs defined by BLL. In the example above, we could get the desired behavior by calling the BLL function `constant()` on the literal string, to delay evaluation.

  - **Minimalism.** Rather than overload a large number of operators and include a large number of primitives, we have chosen a minimalist approach. Thus we have only overloaded four operators for lambda language (array brackets for postfix function application, modulus for infix function application, comma for function argument lists, and equality for "let" assignments). Similarly, apart from `lambda`, the only primitives we provide are those for `let`, `letrec`, and if-then-else expressions. These provide a minimal core of expressive power for lambda, without overburdening the user with a wide interface. A narrow interface seems more likely to be remembered and thus less error-prone.
  - **Visual distinctiveness.** Rather than trying to make lambda expressions "blend in" with normal C++ code, we have done the opposite. We have chosen operators which look big and boxy to make lambda expressions "stand out" from normal C++ code. By convention, we name lambda variables with capital letters. By making lambda expressions visually distinct from normal C++ code, we hope to remind the user which code is "lambda" and which code is "normal C++", so that the user won't accidentally mix the two in ways which create accidents of early evaluation.
- **Capture semantics (lambda-specific).** The FC++ library passes arguments by `const&` throughout the library. Effectively this is just another (perhaps efficient) way of saying "by value". As a result, FC++ lambdas capture free variables by value. As with the rest of the FC++ library, the user can explicitly choose reference semantics by capturing *pointers* to objects, rather than the capturing objects themselves.
- **Compiler error messages.** Meta-programming can be used to detect some user errors and diagnose them "within the library" by injecting *custom error messages*[9, 10] into the compiler output. Though many kinds of errors cannot be caught early by the library (lambdas and functoids can often be passed around in potentially legal contexts, but then finally used deep within some template in the wrong context), there are a number of common types of errors that can be nipped in the bud. The FC++ lambda library catches a number of these types of errors and generates custom error messages for them.
- **Performance.** There seems to be little that we (as library authors) can do here. As expression template libraries continue to become more popular, we can only hope that compilers will become more adept at compiling them quickly. In the meantime, clients of expression template libraries must put up with longer compile times and larger executables.

Thus, given the intrinsic problems/limitations of expression template libraries, we have designed our library to try to minimize those issues whenever possible.

5.4 Lambda in FC++

We now describe what it looks like to do lambda in FC++. Figure 2 shows some examples of lambda. There are a few points which deserve further attention.

Inside lambda, one uses square brackets instead of round ones for postfix functional call. (This works thanks to the lambda-awareness of full functoids, mentioned in Section 4.) Similarly, the percent sign is used instead of the carat for infix function call.

```
// declaring lambda variables
LambdaVar<1> X;
LambdaVar<2> Y;
LambdaVar<3> F;

// basic examples
lambda(X,Y)[ minus[Y,X] ]       // flip(minus)
lambda(X)[ minus[X,3] ]         // minus(_,3)

// infix syntax
lambda(X,Y)[ negate[ 3 %multiplies% X ] %plus% Y ]

// let
lambda(X)[ let[ Y == X %plus% 3,
                F == minus[2]
         ].in[ F[Y] ] ]

// if-then-else
lambda(X)[ if0[ X %less% 10, X, 10 ] ]   // also if1, if2

// letrec
lambda(X)[letrec[F==lambda(Y)[if1[Y %equal% 0,
                                  1,
                                  Y %multiplies% F[Y%minus%1] ]
         ].in[ F[X] ] ]    // factorial
```

Fig. 2. Lambda in FC++

These symbols make lambda code visually distinct so that the appearance of normal-looking (and thus potentially erroneous) code inside a lambda will stand out. Since `operator[]` takes only one argument in C++, we overload the comma operator to simulate multiple arguments. Occassionally this can cause an early evaluation problem, as seen in the code here:

```
// assume f takes 3 integer arguments
lambda(X)[ f[1,2,X] ]   // oops! comma expression "1,2,X"
                        // means "2,X"
lambda(X)[ f[1][2][X] ] // ok; use currying to avoid the issue
```

Unfortunately, C++ sees the expression “`1,2`” and evaluates it eagerly as a comma expression on integers.[2] Fortunately, there is a simple solution: since all full functoids are curryable, we can use currying to avoid comma. The issues with comma suggest another problem, though: how do we call a zero-argument function inside lambda? We found no pretty solution, and ended up inventing this syntax:

```
// assume g takes no arguments and returns an int
// lambda(X)[ X %plus% g[] ]   // illegal: g[] doesn’t parse
lambda(X)[ X %plus% g[_*_] ]   // _*_ means "no argument here"
```

It’s better to have an ugly solution than none at all.

[2] Some C++ compilers, like g++, will provide a useful warning diagnostic (“left-hand-side of comma expression has no effect”), alerting the user to the problem.

The if-then-else construct deserves discussion, as we provide three versions: `if0`, `if1`, and `if2`. `if0` is the typical version, and can be used in most instances. It checks to make sure that its second and third arguments (the “then” branch and the “else” branch) will have the same type when evaluated (and issues a helpful custom error message if they won’t). The other two ifs are used for difficult type-inferencing issues that come from `letrec`. In the factorial example at the end of Figure 2, for example, the “else” branch is too difficult for FC++ to predict the type of, owing to the recursive call to `F`. This results in `if0` generating an error. Thus we have `if1` and `if2` to deal with situations like these: `if1` works like `if0`, but just assumes the expression’s type will be the same as the type of the “then” part, whereas `if2` assumes the type is that of the “else” part. In the factorial example, `if1` is used, and thus the “then” branch (the `int` value `1`) is used to predict that the type of the whole `if1` expression will be `int`.

Having three different ifs makes the lambda interface a little more complicated, but the alternatives seemed to be either (1) to dispose of custom error messages diagnosing if-then-elses whose branches had different types, or (2) to write meta-programs to solve the recursive type equations created by `letrec` to figure out its type within the library. Option (1) is unattractive because the compiler-generated errors from non-parallel if-then-elses are hideous, and option (2) would greatly complicate the metaprogramming in the library and slow down compile-times even more. Thus we think our design choice is justified. Of course, in the vast majority of cases, `if0` is sufficient and this whole issue is moot; only code which uses `letrec` may need `if1` or `if2`.

5.5 Naming the C++ types of lambda expressions

Expression templates often yield objects with complex type names, and FC++ lambdas are no different. For example, the C++ type of

```
// assume:  LambdaVar<1> X;  LambdaVar<2> Y;
lambda(X,Y)[ (3 %multiplies% X) %plus% Y ]
```

is

```
fcpp::Full2<fcpp::fcpp_lambda::Lambda2<fcpp::fcpp_lambda::exp::
Call<fcpp::fcpp_lambda::exp::Call<fcpp::fcpp_lambda::exp::Value<
fcpp::Full2<fcpp::impl::XPlus> >,fcpp::fcpp_lambda::exp::CONS<
fcpp::fcpp_lambda::exp::Call<fcpp::fcpp_lambda::exp::Call<fcpp::
fcpp_lambda::exp::Value<fcpp::Full2<fcpp::impl::XMultiplies> >,
fcpp::fcpp_lambda::exp::CONS<fcpp::fcpp_lambda::exp::Value<int>,
fcpp::fcpp_lambda::exp::NIL> >,fcpp::fcpp_lambda::exp::CONS<fcpp
::fcpp_lambda::exp::LambdaVar<1>,fcpp::fcpp_lambda::exp::NIL> >,
fcpp::fcpp_lambda::exp::NIL> >,fcpp::fcpp_lambda::exp::CONS<fcpp
::fcpp_lambda::exp::LambdaVar<2>,fcpp::fcpp_lambda::exp::NIL> >,1,2> >
```

In the vast majority of cases, the user never needs to name the type of a lambda, since usually the lambda is just being passed off to another template function. Occasionally, however, you want to store a lambda in a temporary variable or return it from a function, and in these cases, you’ll need to name its type. For those cases, we have designed the `LEType` type computer, which provides a way to name the type of a lambda expression (LE). In the example above, the type of

```
lambda(X,Y)[ (3 %multiplies% X) %plus% Y ]
// desugared: lambda(X,Y)[ plus[ multiplies[3][X] ][Y] ]
```

is

```
LEType< LAM< LV<1>, LV<2>,
CALL<CALL<Plus,CALL<CALL<Multiplies,int>,LV<1> > >,LV<2> > > >::Type
```

The general idea is that

```
LEType< Translated_LambdaExp >::Type
```

names the type of `LambdaExp`. Each of our primitive constructs in lambda has a corresponding translated version understood by `LEType`:

`CALL`	`[] (function call)`
`LV`	`LambdaVar`
`IF0,IF1,IF2`	`if0[],if1[],if2[]`
`LAM`	`lambda()[]`
`LET`	`let[].in[]`
`LETREC`	`letrec[].in[]`
`BIND`	`LambdaVar == value`

With `LEType`, the task of naming the type of a lambda expression is still onerous, but `LEType` at least makes it possible. Without the `LEType` type computer, the type of lambda expressions could only be named by examining the library implementation, which may change from version to version. `LEType` guarantees a consistent interface for naming the types of lambda expressions.

Finally, it should be noted that if the lambda only needs to be used monomorphically, it is far simpler (though potentially less efficient) to just use an indirect functoid:

```
// Can name the monomorphic "(int,int)->int" functoid type easily:
Fun2<int,int,int> f = lambda(X,Y)[ (3 %multiplies% X) %plus% Y ];
```

5.6 Comparison to other lambda libraries

Here we briefly compare our approach to implementing lambda to that of the other major lambda libraries for C++: the Boost Lambda Library (BLL)[5] and FACT![13].[3]

Boost Lambda Library Whereas FC++ takes the minimalist approach, BLL takes the maximal approach. Practically every overloadable operator is supported within lambda expressions, and the library has special lambda-expression constructs which mimic the control constructs of C++ (like while loops, switches, exception handling, etc). The library also supports making references to variables, and side-effecting operators like increment and assignment. Lambda is implicit rather than explicit; a reference to a placeholder variables (like `_1`) turns an expression into a lambda on-the-fly.

[3] The FACT! library, like FC++, includes features other than lambda, e.g. functions like `map()` and `foldl()` as well as data structures for lazy evaluation. BLL, on the other hand, is concerned only with lambda.

BLL's approach makes sense given the "target audience"; the Boost libraries are designed for everyday C++ programmers. These are people who are familiar with C++ constructs, and who are hopefully C++-savvy enough to avoid most of the pitfalls of an expression-template lambda library. In contrast, FC++ is designed to support functional programming in the style of languages like Haskell. A number of our users come from other-language backgrounds, and aren't too familiar with the intricacies of C++. Thus FC++'s lambda is designed to present a simple interface with syntax and constructs familiar to functional programmers, and to shield users from C++-complexities as much as possible.

FACT! FACT!, like FC++, is designed to support pure functional programming constructs. Lambda expressions always perform capture "by value" and the resulting functions are typically effect-free. Like FC++, FACT! has an explicit lambda construct; the user can define his own names for placeholder variables, but conventionally names like `x` and `y` are used. FACT! defines few primitive control constructs in its lambda sublanguage (just `where` for if-then-else). Like BLL, however, FACT! overloads many C++ operators (like `+`) for use in lambda expressions. Thus FACT!'s interface is relatively simple and minimal, but lambda expressions are not as visually distinctive as they are in FC++.

6 Monads

Monads provide a useful way to structure programs in a pure functional language. Using monads, it is relatively straightforward to implement things like global state, exceptions, I/O, and other concepts common to impure languages that are otherwise difficult to implement in pure functional languages[6, 14].

Supporting monads in FC++ is an interesting task for a number of reasons:

- Many interesting functional programs and libraries use monads; monad support in FC++ makes it easier to port these libraries to C++.
- Monads in Haskell take advantage of some of that language's most expressively powerful syntax and constructs, including *type classes*, *do-notation*, and *comprehensions*. Modelling these in C++ helps us better understand the relationship between the expressive power of these languages.
- Monads provide a way to factor out some cross-cutting concerns, so that local program changes can have global effects. (We discuss a few example applications that illustrate this.)

In the next subsection, we give a short introduction to monadic programming in Haskell. Next we discuss the relationship between *type classes* in Haskell and *concepts* in C++; understanding this relationship facilitates the discussion in the rest of this section. Then we discuss how we have implemented monads in FC++. We end with some example applications of monads.

6.1 Introduction to monads in Haskell

We briefly introduce a small portion of the Haskell programming language,[4] as its type system provides perhaps the most succinct and transparent way to understand the details of what a monad is. For the moment, know that a monad is a particular kind of data type, which supports two operations (named `unit` and `bind`) with certain signatures that obey certain properties. We shall return to the details after a short digression with Haskell.

In Haskell, the declaration `o :: T` says that object `o` has type `T`. Basic type names (like `Int`) start with capital letters. Lowercase letters are used for free type variables (parametric polymorphism – e.g. templates). The symbol `[T]` represents a list of `T` objects. The symbol `->` separates function arguments and results. The symbol `--` starts a comment. Here are a few examples.

```
x    :: Int          -- x is an integer

add1 :: Int -> Int   -- add1 is a function from Int to Int

-- plus takes two Ints and returns an Int
-- (Or, equivalently, plus takes one Int, and returns a function
-- which takes an Int and returns an Int. Currying is built in.)
plus :: Int -> Int -> Int

-- id takes any type of object and returns
-- an object of the same type
id   :: a -> a

-- map is a polymorphic function of two arguments;
-- it takes a function from type a to type b, and a
-- list of objects of type a, and returns a list of b objects
map  :: (a -> b) -> [a] -> [b]
```

Free type variables can be bounded by “type classes” (described shortly). For example, a function to sort a list requires that the type of elements in the list are comparable with the less-than operator. In Haskell we would say:

```
sort :: (Ord a) => [a] -> [a]
```

That is, `sort` is a function which takes a list of `a` objects and returns a list of `a` objects, subject to the constraint that the type `a` is a member of the `Ord` type class. Type class `Ord` in Haskell represents those types which support ordering operators like

```
class Ord a where
   ==  :: a -> a -> Bool
   <   :: a -> a -> Bool
   <=  :: a -> a -> Bool
      -- etc.
```

[4] Haskell programmers will note that we are fudging some of the details of Haskell to simplify the discussion.

We say that a type `T` is an *instance* of type class `C` when the type supports the methods in the type class. For example, it is true that

```
   instance Ord Int     -- Int is an instance of Ord
```

Given this overview of Haskell's types and type classes, we can now describe monads. A monad is a type class with two operations:

```
 class Monad m where
    bind :: m a -> ( a -> m b ) -> m b
    unit :: a -> m a
```

In this case, instances of monads are not types, but rather they are "type constructors". These are like template classes in C++; an example is a list. In C++ `std::list` is not a type, but `std::list<int>` is. The same holds for Haskell; `[]` is not a type, but `[Int]` is. In the code describing the monad type class above, `m` is a type constructor.

It turns out that *lists* are instances of monads:

```
 instance  Monad []  where
     bind m k       = concat (map k m) -- don't worry about these
     unit x         = [x]              -- definitions yet
 -- in the list monad
 -- bind :: [a] -> ( a -> [b] ) -> [b]
 -- unit :: a -> [a]
```

As another example, consider the `Maybe` type constructor. The type "`Maybe a`" represents a value which is either just an `a` object, or else nothing. In Haskell:

```
   data  Maybe a  =  Nothing | Just a

   -- Examples of variables
   x  :: Maybe Int
   x  =  Just 3

   y  :: Maybe Int
   y  =  Nothing
```

`Maybe` also forms a monad with this definition:

```
 instance  Monad Maybe  where
     bind (Just x) k   =  k x       -- don't worry about
     bind Nothing  k   =  Nothing   -- these definitions
     unit x            =  Just x    -- yet
 -- in the Maybe monad
 -- bind :: Maybe a -> ( a -> Maybe b ) -> Maybe b
 -- unit :: a -> Maybe a
```

A refinement of the `Monad` type class is `MonadWithZero`:

```
   class (Monad m) => MonadWithZero m where
      zero :: m a
```

The `zero` element of a monad is a value which is in the monad regardless of what type was passed to the monad type constructor. For lists, the empty list (`[]`) is the `zero`. For `Maybe`, the `zero` is `Nothing`. Not all monads have zeroes, which is why `MonadWithZero` is a separate type class.

Monads with zeroes can be used in *comprehensions* with *guards*. Comprehensions are a special notation for expressing computations in a monad. Haskell supports comprehensions for the list monad; an example is

```
[ x+y | x <- [1,2,3], y <- [2,3], x<y ]
-- results in [3,4,5]
```

This list comprehension could be interpreted as "the list of values x plus y, for all x and y where x is selected from the list [1,2,3] and y is selected from the list [2,3], and where x is less than y". The desugared version of the Haskell code is:

```
-- (\z -> z+1) is Haskell lambda syntax:
--    (lambda(Z)[ Z %plus% 1 ])
-- backquotes are Haskell's infix syntax:
--    (x ‘f‘ y == f x y)
[1,2,3] ‘bind‘ (\x ->
   [2,3]   ‘bind‘ (\y ->
      if not (x<y) then zero
                   else unit (x+y) ))
```

The translation from the comprehension notation to the desugared code is straightforward. Starting from the vertical bar and going to the right, the expressions of the form "`var <- exp`" turn into calls to `bind` and lambdas, and guards (boolean conditions) are transformed into if-then-else expressions which return the monad `zero` if the condition fails to hold. After all expressions to the right of the vertical bar have been processed, the expression to the left of the vertical bar gets `unit` called on it to lift the final computed value back into the monad.

6.2 Haskell's type classes and C++ template concepts

In the C++ literature, we sometimes speak of template *concepts*. A concept in C++ is a set of constraints which a type is required to meet in order to be used to instantiate a template. For example, in the implementation of the template function `std::find()`, there will undoubtedly be some code along the lines of

```
if( cur_element == target ) // ...
```

which compares two elements for equality using the equality operator. Thus, in order to call `std::find()` to find a value in a container, the element type must be `EqualityComparable`—that is, it must support the equality operator with the right semantics. We call `EqualityComparable` a *concept*, and we say that types (such as `int`) which meet the constraints *model* the concept. Concepts exist only implicitly in the C++ code (e.g. owing to the call to `operator==()` in the implementation), and often exist explicitly in documentation of the library. Some C++ libraries[9, 10] are devoted to "concept checking", these libraries check to see that the types used to instantiate a template do indeed model the required concepts (and issue a useful error message if not).

Haskell type classes are analogous to C++ concepts. However in Haskell they are reified; there are language constructs to define type classes and to declare which types are instances of those type classes. In C++, when a certain type models a certain concept (by meeting all of the appropriate constaints), it is merely happenstance (structural conformance); in Haskell, however, in addition to meeting the constraints of a type class interface, a type must be declared to be an instance of the concept (named conformance). “Concept checking” in Haskell is built into the language: type classes define concepts, instance declarations say which types model which concepts, and type bounds make explicit the constraints on any particular polymorphic function.

Understanding this analogy will make the FC++ implementation of monads more transparent. As we shall see, in the FC++ library, we spell out the concept requirements on monads, in order to make it easier for clients who write monads to ensure that they have provided all of the necessary functionality in the templates.

6.3 Comparing monads in FC++ to those in Haskell

```
 struct AUniqueTypeForNothing ;
 AUniqueTypeForNothing NOTHING;

 template <class T>
 class Maybe
    List<T> rep;
 public:
    typedef T ElementType;

    Maybe( AUniqueTypeForNothing )
    Maybe()                               // Nothing constructor
    Maybe(const T& x) : rep(cons(x,NIL))  // Just constructor

    bool is_nothing() const  return null(rep);
    T value() const  return head(rep);
 ;

 struct XJust
    template <class T>
struct Sig : public FunType<T,Maybe<T> > ;

    template <class T>
    typename Sig<T>::ResultType
    operator()( const T& x ) const
       return Maybe<T>( x );

 ;
 typedef Full1<XJust> Just;
 Just just;
```

Fig. 3. The `Maybe` datatype in FC++

Let us now illustrate monad definitions in FC++. As a first example, we shall look at `Maybe`. The `Maybe` template class and its associated entities are defined in Figure 3. `NOTHING` is the constant which represents an “empty” `Maybe`, and `just()` is a functoid which turns a value of type `T` into a “full” `Maybe<T>`. (`Maybe` is implemented using a `List` which holds either one or zero elements.)

```
/*
 concept Monad
   // full functoid with Sig  unit :: a -> m a
   typedef Unit;
   static Unit unit;
   // full functoid with Sig  bind :: m a -> ( a -> m b ) -> m b
   typedef Bind;
   static Bind bind;

 concept MonadWithZero models Monad
   // zero :: m a
   typedef Zero;        // a value type
   static Zero zero;

*/
```

Fig. 4. Documentation of the monad concept requirements in FC++

Next we consider how to make `Maybe` a monad. Figure 4 describes the general monad concepts as specified in the FC++ documentation. A monad class must define the methods `unit` and `bind` (with the appropriate signatures); a class representing a monad with a zero must meet the above requirements as well as defining a `zero` element.

Figure 5 shows how we define the `Maybe` monad in FC++. Nested in `struct MaybeM` we define `unit`, `bind`, and `zero`, as well as `typedef`s for their types. This FC++ definition effectively corresponds to the definitions

```
 instance Monad Maybe -- ...
 instance MonadWithZero Maybe -- ...
```

in Haskell.

It should be noted here that the one major difference between monads in FC++ and monads in Haskell is that, in FC++, there is a distinction between the monad type constructor (e.g. `Maybe`) and the monad itself (e.g. `MaybeM`). We chose to make this distinction for reasons discussed next.

One advantage to separating the type constructor (`Maybe`) from the monad definition (`MaybeM`) is that, since the monad definition is itself a data type, it can be used as a type parameter to template functions. As a result, rather than supporting just list comprehensions (like Haskell does), in FC++ we support *comprehensions in an arbitrary monad*, by passing the monad as a template parameter to the comprehension. For example, the Haskell list comprehension

```
 [ x+y | x <- [1,2,3], y <- [2,3], x<y ]
```

```
struct MaybeM
   typedef Just Unit;
   static Unit unit;

   struct XBind
      template <class M, class K> struct Sig : public FunType<M,K,
         typename RT<K,typename M::ElementType>::ResultType> ;
      template <class M, class K>
      typename Sig<M,K>::ResultType
      operator()( const M& m, const K& k ) const
         if( m.is_nothing() )
            return NOTHING;
         else
            return k( m.value() );

   ;
   typedef Full2<XBind> Bind;
   static Bind bind;

   typedef AUniqueTypeForNothing Zero;
   static Zero zero;
;
```

Fig. 5. Definition of the `Maybe` monad (`MaybeM`)

is written in FC++ as

```
compM<ListM>()[ X %plus% Y |
   X <= list_with(1,2,3), Y <= list_with(2,3), guard[ X %less% Y ] ]
```

Note how `ListM` is passed as an explicit template parameter to the `compM` function, which returns a comprehension for that monad. As a result, we can write

```
compM<MaybeM>()[X %plus% Y | X <= just(2), Y<=just(3)]
```

and perform a comprehension in the `Maybe` monad. Having a name apart from the data type constructor to serve as a handle for the monad definition (e.g. `ListM`, `MaybeM`) gives us a convenient way to parameterize monad operations. (The idea of generalizing comprehensions to arbitrary monads was originally discussed by Wadler[15].)

There is another advantage to separating the type constructor from the monad definition. Haskell type classes require algebraic data type constructors (not type aliases) to work. As a result, we cannot express the identity monad (a monad where `m a = a`) directly in Haskell. Instead we have to fake it by defining a new data type (which we have chosen to call `Identity`):

```
data Identity a = Ident a

instance Monad Identity where    --  m a = Identity a
   unit x   = x
   bind m k = k m
```

where values of type `a` are wrapped/unwrapped with the value constructor `Ident` to make them members of the type `Identity a`. In FC++, however, we can define the monad without also having to define a new data type to represent identities, as seen in Figure 6. The reason for the distinction is perhaps obvious. Haskell uses type inference, which means it must unambiguously be able to figure out which monad a particular data type is in. This type inference is not possible unless there is a one-to-one mapping between algebraic datatype constructors and monads. In FC++, on the other hand, the user passes the monad explicitly as a template parameter to constructs like `compM`. By requiring the user to be a little more explicit about the types, we gain a bit of expressive freedom (e.g. being able to do comprehensions in arbitrary monads).

```
// Nothing corresponding to Identity data type needed by Haskell
struct IdentityM     // M a = a
   typedef Id Unit;
   static Unit unit;

   struct XBind
      template <class M, class K> struct Sig : public
         FunType<M,K, typename RT<K,M>::ResultType> ;
      template <class M, class K>
      typename Sig<M,K>::ResultType
      operator()( const M& m, const K& k ) const
         return k(m);

   ;
   typedef Full2<XBind> Bind;
   static Bind bind;
;
```

Fig. 6. Definition of the `IdentityM` monad

6.4 Monads in FC++

The previous subsection introduced FC++ monads. Here we flesh out exactly what monad support FC++ provides.

FC++ provides functoids for the main monad operations. Specifically:

```
unitM<SomeMonad>()  // SomeMonad's "unit" functoid
bindM<SomeMonad>()  // SomeMonad's "bind" functoid
zeroM<SomeMonad>()  // SomeMonad's "zero" value
plusM<SomeMonad>()  // SomeMonad's "plus" functoid
bindM_<SomeMonad>() // SomeMonad's "bind_" functoid
mapM<SomeMonad>()   // SomeMonad's "map" functoid
joinM<SomeMonad>()  // SomeMonad's "join" functoid
liftM<SomeMonad>()  // lifts a one-arg function into SomeMonad
liftM2<SomeMonad>() // lifts a two-arg function into SomeMonad
liftM3<SomeMonad>() // lifts a three-arg function into SomeMonad
bind                // "bind" (monad is inferred)
bind_               // "bind_" (monad is inferred)
```

Many of these have not been previously mentioned; `plusM` is another function supported by some monads; `bindM_`, `mapM`, `joinM`, and the `liftM` functions are common monad operations which are defined in terms of unitM and bindM; bind and bind_ are described more below.

FC++ supports comprehensions in arbitrary monads, using the general syntax:

```
compM<SomeMonad>()[ lambdaExp | thing, thing, ... thing ]
```

where thing is one of

– a gets expression of the form “LV <= lambdaExp” (Translates into a call to bind)
– a lambda expression (Translates into a call to bind_)
– a guard expression of the form “guard[boolLambdaExp]” (Translates into an if-then-else with zero if the test fails)

This is similar to the syntax used by Haskell’s list comprehensions. FC++ also supports a construct similar to Haskell’s *do-notation*:

```
doM[ thing, thing, ... thing ]
```

where each thing is as before, only guards are no longer allowed. (The lack of a monad parameter to doM is discussed shortly.)

Clients can define monads by creating monad classes which model the monad concepts described in the previous subsection (Monad and MonadWithZero). There is also a MonadWithPlus concept for monads which support plus. Additionally there is another concept called InferrableMonad, which may be modelled when there is a one-to-one correspondence between a datatype and a monad. In the case of InferrableMonads, FC++ (like Haskell) can automatically infer the monad based on the datatype in some cases; constructs like doM and the functoids bind and bind_ do not need to have a monad passed an an explicit parameter—they infer it automatically.

The monad syntax is part of FC++’s lambda sublanguage. As with lambda, we strived for minimalism when implementing monads. The only new operator overloads are operator| and operator<=, and the only new syntax primitives are compM, guard, and doM. As with the rest of lambda, we provide LEType translations so that clients can name the result type of lambda expressions which use monads:

DOM	doM[]
GETS	LambdaVar <= value
GUARD	guard[]
COMP	compM<SomeMonad>()[]

As with the other portions of lambda, FC++ provides some custom error messages for common abuses of the monad constructs. We followed the same design principles discussed in Section 5 when implementing monads in FC++.

6.5 Monad examples

There are many example applications which use monads; here we discuss a small sample to give a feel for what monads are useful for.

Using MaybeM for exceptions One classic example of the utility of monads comes from the domain of exception handling. Suppose we have written some code which computes some values using some functions:

```
   x = f(3);
   y = g(x);
   z = h(x,y);
   return z;
```

(For the sake of argument, let's say that the functions `f`, `g`, and `h` take positive integers as arguments and return positive integers as results.) Now suppose that each of the functions above may fail for some reason. In a language with exceptions, we could throw exceptions in the case of failure. However in a language without an exception mechanism (like C or Haskell), we would typically be forced to represent failure using some sentinel value (`-1`, say), and then change the client code to

```
x = f(3);
if( x == -1 ) {
   return -1;
} else {
   y = g(x);
   if( y == -1 ) {
      return -1;
   } else {
      z = h(x,y);
      return z;
   }
}
```

This is painful because the "exception handling" part of the code clutters up the main line code. However, we can solve the problem much more simply by using the Maybe monad. Let the functions return values of type `Maybe<int>`, and let `NOTHING` represent failure. Now the client code can be written as just

```
compM<MaybeM>()[ Z | X <= f[3],
                     Y <= g[X],
                     Z <= h[X,Y] ]
```

The definitions of `unit` and `bind` in the `MaybeM` monad make the problem trivial; `NOTHING` values immediately propogate up through the end of the comprehension, whereas integers continue on through the computation as desired.

Using ListM for non-determinism Now imagine changing the problem above slightly; instead of the functions `f`, `g`, and `h` having the possibility of failure, suppose instead that they are non-deterministic. That is, suppose each function returns not a single integer, but rather a list of all possible integer results. Changing the original client code to deal with this change would likely be even uglier than the original change (which required all the tests for `-1`). However the change to the monadic version is trivial:

```
compM<ListM>()[ Z | X <= f[3],  -- Note ListM instead of MaybeM
                    Y <= g[X],
                    Z <= h[X,Y] ]
```

The result is a list of all the possible integer values for `Z` which satistfy the formulae.

A monadic evaluator Wadler [15] demonstrates the utility of monads in the context of writing an expression evaluator. Wadler gives an example of an interpreter for a tiny expression language, and shows how adding various kinds of functionality, such as error handling, counting the number of reduction operations performed, keeping an execution trace, etc. takes a bit of work. The evaluator is then rewritten using monads, and the various additions are revisited. In the monadic version, the changes necessary to effect each of the additions are much smaller and more local than the changes to the original (non-monadic) program. This example demonstrates the value of using monads to structure programs in order to localize the changes necessary to make a wide variety of additions throughout a program.

Monadic parser combinators Parsing is a domain which is especially well-suited to monads. In the Haskell community, "monadic parser combinators" are becoming the standard way to structure parsing libraries. As it turns out, parsers can be expressed as a monad: a typical representation type for parser monads is

```
Parser a = String -> Maybe ( a, String )  -- the monad "Parser"
```

That is, a parser is a function which takes a `String` and returns

- (if the parse succeeds) a pair containing the result of the parse and the remaining (yet unparsed) `String`, or
- (if the parse fails) `Nothing`.

Monadic parser *combinators* are functions which combine parsers to yield new parsers, typically in ways commonly found in the domain of parsing and grammars. For example, the parser combinator `many`:

```
many :: Parser a -> Parser [a]
```

implements Kleene star—for example, given a parser which parses a single digit called "`digit`", the parser "`many digit`" parses any number of digits. Monadic parser combinator libraries typically provide a number of basic parsers (e.g. `charP`, which parses any character and returns that character) and combinators (e.g. `plusP`, which takes two parsers and returns a new parser which tries to parse a string with the first parser, but if that fails, uses the second) to clients. The beauty of the monadic parser combinator approach is that it is easy for clients to define their own parsers and combinators for their specific needs. A good introductory paper on the topic of monadic parser combinators in Haskell is [3]; we implement the examples in that paper in one of the example files that comes with the FC++ library.

As we have seen in the previous examples, using monads often makes it easy to change some fundamental aspect of the behavior of the program. For example, if we have an ambiguous grammar (one for which some strings admit multiple parses), we can simply change the representation type for the parser like so:

```
  Parser a = String -> [ ( a, String ) ]
-- uses List instead of Maybe
```

and redefine the monad operations (unit, bind, zero, and plus), and then parsers will return a list of every possible parse of the string. This is all possible without making any changes to existing client code.

One alternative approach to writing parsing libraries in C++ is that taken by the Boost Spirit Library[1]. Spirit uses expression templates to turn C++ into a yacc-like tool, where parsers can be expressed using syntax similar to the language grammar. For example, given the expression language

```
factor      ::= integer | group           // BNF
term        ::= factor (mulOp factor)*
expression  ::= term (addOp term)*
group       ::= '(' expression ')'
```

one can write a parser using Spirit as

```
factor      = integer | group;                // Spirit (C++)
term        = factor >> *(mulOp >> factor);
expression  = term >> *(addOp >> term);
group       = '(' >> expression >> ')';
```

which is almost just as readable as the grammar. Like yacc, Spirit has a way to associate semantic actions with each rule.

The results are similar with monadic parser combinators. Using an FC++ monadic parser combinator library, we can write

```
factor     = lambda(S)[(integer %plusP% dereference[&group])[S]];
term       = factor ^chainl1^ mulOp;
expression = term ^chainl1^ addOp;
group      = bracket( charP('('), expression, charP(')') );
```

to express the same parser. The above FC++ code creates parser functoids by using more primitive parsers and combining them with appropriate parser combinators like chainl1. (Note that, whereas Spirit's parser rules are effectively "by reference", FC++ functoids are "by value", which means we need to explicitly create indirection to break the recursion among these functoids. Hence the use of lambda, dereference, and the address-of operator.) This FC++ parser not only parses the string, but it also evaluates the arithmetic expression parsed. The semantics are built into the user-defined combinators like addOp and chainl1. For example,

```
addOp :: Parser (Int -> Int -> Int)
```

parses a symbol like '-' and returns the corresponding functoid (minus). Then,

```
chainl1 :: Parser a -> Parser (a -> a -> a ) -> Parser a
-- e.g.   p ‘chainl1‘ op
```

parses repeated applications of parser p , separated by applications of parser op (whose result is a left-assocative function, which is used to combine the results from the p parsers). Thus monadic parser combinator libraries allow one to express parsers at a level of abstraction comparable to tools like yacc or the Spirit library, but in a way in which users can define their own abstractions (like chainl1) for parsing and semantics, rather than just using the builtin ones (like Kleene star) supplied by the tool/library.

Lazy evaluation Previous versions of FC++ supported lazy evaluation in two main ways: first, via the lazy `List` class and the functions (like `map`) that use `List`s, and second, via "thunks" (zero argument functoids, like `Fun0<T>`). Monads provide a new, more general mechanism to lazify computations. The datatype `ByNeed<T>` and its associated monad `ByNeedM` can be used to make a computation lazy. Additionally, the functoid `bLift` lazifies a functoid by lifting its result into the ByNeedM monad. For example, we can lazify

```
x = f(3);
y = g(x);
z = h(x,y);
```

by writing

```
compM<ByNeedM>()[ Z | X <= bLift[f] [3],
                      Y <= bLift[g] [X],
                      Z <= bLift[h] [X,Y] ]
```

The result is a `ByNeed<int>` value, which is a computation that will result in an `int` when "forced" by calling `bForce`. (Conversely, a constant can be turned into a by-need computation by calling `bDelay`.) Using values of type `ByNeed<T>` in lieu of type `T` ensures that lazy evaluation occurs: a computation is not performed until the value is demanded, and once a computation has been run to produce a value, the value is cached so that further applications of `bForce` get the cached value rather than re-running the computation.

In short, the datatype `ByNeed<T>` combines "thunks" with caching, and the `ByNeedM` monad makes syntax sugar like comprehensions available so that client code working with `ByNeed<T>` objects need not be concerned with all the "forcing" and "delaying" in the midst of the computation (the monad plumbing handles this).

Summary The examples given in this section give a sense of the kinds of applications for which monads are useful. Monads have a wide variety of utilities, which span varied domains (such as parsing and lists) and a number of cross-cutting concerns (like lazy evaluation and exception handling). Prior versions of FC++ implemented a few small monads, but they were extremely burdensome to express. The expressiveness afforded by the new FC++ syntactic sugar (like lambda and comprehensions) makes using monads in C++ a practicality for the first time.

7 Conclusions

We have given an overview of FC++ and described its new features in detail. Full functoids provide a general and reusable mechanism for adding features such as curryability, infix syntax, and lambda-awareness to every functoid. The lambda sublanguage is designed to minimize the problems common to all expression-template lambda libraries in C++. We have discussed the relationship between Haskell type classes and C++ template concepts in order to help describe how monads can be expressed in FC++. We have demonstrated a novel syntax for comprehensions which generalizes this construct to an arbitrary monad. Throughout FC++ and the lambda sublanguage, we have overloaded a select few operators to provide syntactic sugar for the library and we have used named functoids like `plus` to express the actual operations of C++ operators.

Bibliography

[1] de Guzman, Joel, et al. The Boost Spirit Library. Available at `http://www.boost.org/libs/spirit/index.html`

[2] *Haskell 98 Language Report*. Available online at `http://www.haskell.org/onlinereport/`

[3] Hutton Graham and Meijer Erik. "Monadic parsing in Haskell" *Journal of Functional Programming*, 8(4):437-444, Cambridge University Press, July 1998.

[4] *ISO/IEC 14882: Programming Languages – C++*. ANSI, 1998.

[5] Järvi, Jaakko and Powell, Gary. The Boost Lambda Library. Available at `http://boost.org/libs/lambda/doc/index.html`

[6] Jones, Simon Peyton and Wadler, Philip. "Imperative functional programming," *20th Symposium on Principles of Programming Languages*, ACM Press, Charlotte, North Carolina, January 1993.

[7] McNamara, Brian and Smaragdakis, Yannis. "FC++: Functional Programming in C++", *Proc. International Conference on Functional Programming (ICFP)*, Montreal, Canada, September 2000.

[8] McNamara, Brian and Smaragdakis, Yannis. "Functional Programming with the FC++ library" *Journal of Functional Programming*, to appear.

[9] McNamara, Brian and Smaragdakis, Yannis. "Static Interfaces in C++" *Workshop on C++ Template Programming* October 2000, Erfurt, Germany. Available at `http://www.oonumerics.org/tmpw00/`

[10] Siek, Jeremy and Lumsdaine, Andrew. "Concept Checking: Binding Parametric Polymorphism in C++" *Workshop on C++ Template Programming* October 2000, Erfurt, Germany. Available at `http://www.oonumerics.org/tmpw00/`

[11] Y. Smaragdakis and B. McNamara, "FC++: Functional Tools for Object-Oriented Tasks" *Software Practice and Experience*, August 2002.

[12] A. Stepanov and M. Lee, "The Standard Template Library", 1995. Incorporated in ANSI/ISO Committee C++ Standard.

[13] Striegnitz, Jörg. "FACT! The Functional Side of C++," Available at `http://www.fz-juelich.de/zam/FACT`

[14] Wadler, Philip. "Comprehending monads," *Mathematical Structures in Computer Science*, Special issue of selected papers from 6th Conference on Lisp and Functional Programming, 2:461-493, 1992.

[15] Wadler, Philip. "Monads for functional programming." J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.

Importing alternative paradigms into modern object-oriented languages.

Andrey V. Stolyarov

Moscow State Lomonosov University,
dept. of Computational Math. and Cybernetics,
MGU, II uch. korp., komn.747, Leninskie Gory,
Moscow, 119899, Russia

Abstract. The paper is devoted to the problem of importing alternative paradigms into an imperative object-orented environment. Several known solutions of the problem are discussed with explanation of their drawbacks. Then a new solution is introduced.

The solution is based on the fact that programming paradigms developed within alternative languages such as Lisp, Prolog, Refal etc. are in fact independent from their respective languages (e.g., from their syntax). Each of these languages implements a certain algebra, which in fact creates the paradigms. It is possible to represent such an algebra with object-oriented technique and get the respective set of paradigms within the primary language. Together with syntactic capabilities of the primary language (such as overloading of standard arythmetic operation symbols) this results in possibility for a programmer to use alternative paradigms (such as Lisp programming) right within the primary language (C++ or Ada95). No changes to the primary language is needed, nor is it required to apply any additional preprocessing to the code; only the standard translator of the primary language is used. The only thing needed to use the explained approach is an appropriate library.

As an illustration, the paper describes a C++ class library named InteLib which currently has a practically usable implementation for Lisp programming, and experimental umplementations of Refal and a subset of Prolog.

1 Introduction

Different programming languages encourage a programmer to use different ways to imagine the program being developed, the environment (hardware, operating system, user etc) and their interaction. The simplest way is to imagine a computer either as such (processor, memory, i/o ports etc) or a some kind of virtual machine capable to perform certain set of operations, and a program as a sequence of instructions those explicitly specify what operations are to be performed. This way of thinking is known as ”imperative programming”.

Another technique, proposed in early 1960s [10], is to represent the program as a set of functions. Each function gets zero or more arguments and computes a result. A function may use other functions in computations, including using itself, directly or indirectly (so called recursive function calls). No side effects are allowed, that is, if all the arguments are known, one can replace a function call with it’s result and get the same program. This is known as ”pure functional programming” [7].

Several years later, ”logic programming” [12] was proposed. In logic programming, the program is thought as a set of logic facts (axioms) used to test statements and find objects that satisfy the given conditions. Pure logic programming also disallows side effects.

Obviously, both pure functional and pure logic programming are not suitable for interactive programs because interactive program changes its environment at least reading from the input and writing to the output so it’s required to have functions with side effects to create an interactive program. In contrast with these two styles, the technique of object-oriented programming appeared in the middle 1970s looks like being created specially for interactive software development. The program and its environment are represented with so-called ”objects” – abstract ”black boxes” capable to exchange messages and perform various actions in response to a message [2].

The notion of a *programming paradigm* is often used to refer to a particular system of programming abstractions. It is important to notice that the notion of a programming paradigm is significantly unformal. There is no well-established classification of programming paradigms though there were many attempts to give one (e.g., [13]). For example, the languages Lisp[15], Refal[17], Miranda[18] and Hope[7] are all usually taken as functional languages. However, they in fact have less in common than in differences. E.g., a Refal function is based on text matching against patterns and transforming in accordance to the rule for the pattern first matched. There’s no such capability in Lisp. This makes it more convenient to perform lexical and syntactic analyses with Refal than with Lisp. Hope and Miranda are pure functional languages while Lisp has global variables and local lexical bindings changeable as a side effect of a function, etc.

Nowadays the union of imperative and object-oriented paradigms is the most popular in the software industry. The union is implemented by such languages as C++ [16], Ada95, Java, Delphi, Object Pascal etc.

Using different languages together within a single project leads to different serious problems so it is rare practice; most projects are single-language, and the langage is one of these imperative object-oriented languages listed above. In most cases it is inconvenient to implement a whole project in Lisp, Refal or Prolog. This in fact results in that these languages are rarely used at all despite that they are extremely suitable for some subtasks in almost any project.

2 Example of a project suitable for multiparadigm technique

As noted in [6], ”We may encounter application domains which can be modeled best with only one paradigm. But there may be other domains which can be represented more adequately using multiple paradigms”. Furthermore, in almost any large software project there are subtasks for which alternative paradigms are suitable.

Consider we have a database with complicated relationship within its components, and a user who needs a good interface to the database, preferably close to a natural language. First of all, we have to analyse the queries user types at his console (lexical analysis). Then we need to determine what do they mean (syntax analysis). Finally, we need to create and perform a query to the database and return its results to the user.

It takes significant time to write a lexical analyser in an imperative language such as C++. If we do it in Refal, however, the work's complexity reduces by tens of times.

Next, we need to do some more processing to prepare the query. We should determine what formulae do mean and probably perform some transformations, optimizations etc. It is hard to operate with symbolic formulae in C++, but in Lisp all the symbolic transformations are programmed simply.

Now we are ready to request and retrieve a result from the database. It's not a problem when the query is simple; for instance, if we've got a database storing personal data of some people, a request like ”Give me a home phone number of Bob Johnson” after lexical analysis and syntax transformations is not a problem to perform. However, the user might ask for a thing much harder to calculate. For example: ”Find me a female who has graduated in 1997 in the Moscow State University as a computer science person, then got married in 1998 with a male who speaks fluent English and is older than his wife by 3 years”. It is possible to create a database that stores all the necessary data, but a request like this might force us to write a whole program to complete it. Please note that we know the conditions and the only problem is to find the solution which satisfies them all. Logic languages such as Prolog, Datalog etc. are suitable for this purpose [5].

It looks like a good idea to make data flow such as shown at fig. 1

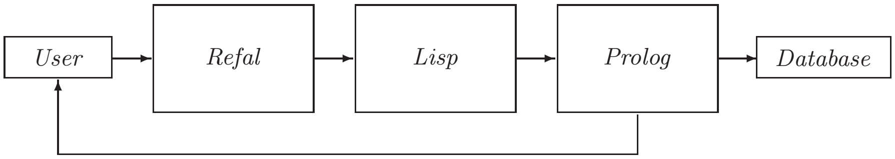

Fig. 1. Simple idea of data flow between parts implemented in different languages

However, the diagram at fig. 1 does not represent all the functionality of the hypotetical system. First, it must interact with a user, possibly via a network. Second, it should control the database, which is probably to be used by many users simultaneously. That leads to many technological problems (e.g., locking) to be solved by the system. In addition, we should not forget that the data is stored on physical (that is, *real*) disks, so we need to monitor the file system, check whether there's sufficient amount of free space, do some caching to increase performance etc.

Languages such as Refal, Prolog and Lisp are not good to do all these things. Their features are far from the real equipment capabilities. They are not so efficient as C++ and other ”universal” imperative languages. For an artificial intelligence tasks such as the one described below, it is possible to trade efficiency of the software for development speed increase, but in a system task envolved to maintain the data storage it is inacceptable to loose in efficiency.

Besides that, interaction with a user in modern systems requires graphical interface (GUI), which is usually created as an event-driven system. It is inconvenient to create an event-driven system with an artificial intelligence language. Object-oriented languages such as SmallTalk[8], C++ or Java are much more suitable for this purpose.

The diagram at fig. 2 is closer to practice. We assume that ”User Interface” and ”Database Management System” are created with languages suitable for these purposes, probably C and C++. So we have C++ as a primary language and Lisp, Prolog and Refal as secondary languages.

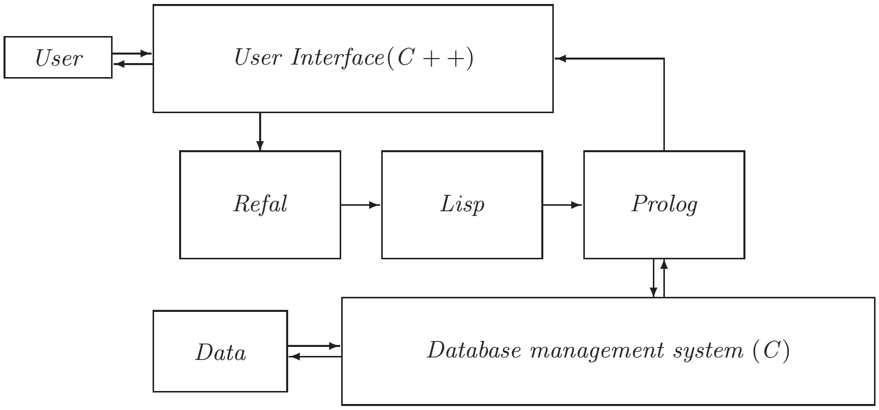

Fig. 2. Data flow closer to reality

However, when someone tries to use such an idea in a real project, she finds out it is much harder to implement such a system than to create a diagram. If we try to use an alternative language for a particular (small) subtask in a large project, we get into a trouble with integration of language tools that have so totally different nature (for instance, strictly typed imperative language as the primary language and a typeless functional language as the secondary language). There are problems in calling conventions, in sharing global data, in using heaps etc.

Furthermore, even the fact of using two or more different programming systems within one project makes the project harder to manage. If one of the programmers does not know one of the used programming systems, she could get into a trouble trying to build the project, to fix someone else’s code etc. The difficulty of managing a project which uses two or more programming systems is so serious that this reason alone is able to prevent senior developers from making decisions of using different languages.

3 Different ways towards multiparadigm environment creation

It is obvious that an ability of using different paradigms together is attractive. There are certain difficulties though that prevent programmers from trying multiparadigm programming.

Before we introduce the new idea which hopefully allows to avoid most of the troubles, let us discuss some possible (and well-known) ways of creation of a multiparadigm environment.

We'll try to understand why each of them didn't become as widely-used as it is necessary to satisfy the need in multiparadigm programming. This will allow us to specify what do we actually want from the new technique.

3.1 Creation of another programming language

There were many attempts to create a new programming language for the purpose of multiparadigm programming (e.g., Leda[3], Oz[11] etc.). There is an unexpected trouble however. It is expensive to develop a new language. It is yet more expensive to bring the newly-developed language to the level of an industrial product so that it can be used in real software engineering practice, because this requires to support the language with useful software tools (compilers, debuggers etc.), as well as to create sufficient amount of documentation, tutorials and write and publish lots of books. But the real trouble is then to wait and see that the software engineering community doesn't tend to use the language in spite of all its advantages.

It is somewhat magic[1] when the community turns towards a new language, and this is a very rare kind of event, probably because the community is too conservative[2]. In fact it has to be. Really, starting to use a new technology requires to reeducate the personnel and change habitual methods of working. Both are very expensive, while outcome of changing languages is not so clear, specially to managers who make decisions.

This is why we decide not to try to create another language, even by extending an existing one. Enough of them are created already but that doesn't help.

3.2 A package of differently implemented programs

There are as well a few approaches to solve the problems of different programming languages integration within a single project. The simplest idea is just to write several programs, each in its own language, and make them interoperate using operating system's capabilities.

This kind of solution avoids problems with calling and data conventions, linking incompatibilities etc. It doesn't help tough with problems of using different programming systems within a single project.

Besides that, interoperation organization produces its own problems depending on a particular technology.

Using Unix style[3], when every program of the package has standard streams of input and output and the interoperation is done with command languages, we have to

[1] E.g., community preferred to use C++ with all its drawbacks while a similar but better-looking and carefully developed language Ada is in fact forgotten

[2] There's nothing bad though in this conservatism. Everything happens too fast in Computer Science so if the industry wasn't so conservative, we'd have nothing actually done.

[3] Unix style is mentioned as a multiparadigm environment in, e.g., [14]

convert all the data somehow into a text representation, and then analyse the representation in another program of the package. Another drawback is that it is not always convenient to use the standard input/output streams for interoperation with another part of the package.

There are some attempts to make another, more convenient for a programmer standard way to organize interoperation of different programs, such as CORBA and COM. However, such technologies theyselves are complicated enough so that making a program to support them could be comparable in difficulty with solving the task the program is actually written for, and they are best handled with object-orientred languages, producing troubles with functional or logic languages.

3.3 Embedded interpreters

Another solution is to build an interpreter of a secondary language into the primary language. In the simplest case the interpreter is implemented as a module, which has an appropriate interface. The main program (e.g., written in C++) feeds the interpreter with the text of a module created in the secondary language (e.g., Lisp), then passes the initial data, runs the interpreter and reads the results back.

One of more advanced techniques is known as ’embeddance’ of one language into another. In this case the primary language is enlarged with some certain constructs which allow to insert constructs of a secondary language into a code in the primary language. In this case we need a preprocessor which handles embeddance constructs and produces a plain code in the primary language which is then compiled in a regular way. In fact such a preprocessor replaces a foreign construct with some code which converts all necessary variables into a text, creates an appropriate query and then passes it to an interpreter, and then converts the received result as necessary. This technique is successfully applied to the case of writing database operation software using SQL-based database management system.

Such a solution, though, has many disadvantages:

- The solution does not allow us to call primary language functions from within the interpreted code. Only primary language functions can call the secondary language code, but not vice versa.
- The secondary language code is fully interpreted. That is, when the program runs, every call to the interpreted language is given in its text representation. The embedded interpreter has to perform lexical and syntactic analyses ”on-the-fly” which may lead to efficiency losses.
- ”The last but not least” – the results of the interpreted code calling are also presented in text form, so we need to analyse it in the main program. Just remember that analysing of strings is one of the tasks we want to avoid in C++ code and perform with an artificial intelligence language, preferably Refal, and you’ll realize that something is wrong.

Besides that, mixing up interpreted and compiled execution within one program doesn’t look like a fair solution anyway. It is clear that we can’t avoid interpretation completely for such languages as Lisp or Prolog, but at least we could expect there will be no lexical and syntactic analyses at runtime.

3.4 Extendable interpreters

The opposite solution is to choose an interpreted language as the primary one and provide mechanisms to extend the interpreter with functions implemented in another language (usually the language in which the interpreter is initially implemented). One of the well-known examples is Tcl. Its interpreter allows to call C code which is compiled into a shared library following certain simple calling conventions.

The main drawback of this method is that the primary language **must** be interpreted which may be inappropriate in some cases.

3.5 Cross-language linkage

Having certain amount of patience, it is possible to compile and link modules written in different languages together. As it was mentioned before, this produces numerous problems with differences in calling conventions, data representation conventions etc. Furthermore, it almost always requires to make changes to the existing programming systems (for example, to reimplement compilers so that they could produce compatible object modules). As of practice, all these hardships are sufficient to prevent programmers from trying multiparadigm programming. And, anyway, we don't reach real integration of languages this way because the languages theyselves are not designed for multilanguage environment (e.g., there's a problem how to call a C function from Lisp code – how to *specify* such a call using Lisp syntax).

3.6 Compilation from one language into another

The difficulties of linking together modules implemented in two defferent languages can be reduced if one of the languages is first compiled into the other.

Many of Scheme translators actually produce C code which can then be compiled in usual manner. Thus there's no problem to link such a code with some modules implemented in C and/or C++.

We still don't know how to specify a C function call in Scheme syntax, that is, only Scheme functions can be called from C, but not vice-versa. Also, the programmer needs to understand the internal data structures of the particular implementation of Scheme in order to compose a call to a Scheme function and/or analyse the results. This is inappropriate because internal data structures are usually not well-documented.

3.7 Paradigms without a language

There's also a chance to brainstorm why do we actually want to use another language for a particular subtask, that is, what features does it have the primary language doesn't provide, and then just implement them (e.g., as a library).

Consider we use C++ as the primary language and we for some reason we feel it useful to have heterogenous lists[4] as we do in Lisp. It is possible to implement them with C++ template classes. First, we create a base class which implements the common

[4] Each element of the list may have its own type

behaviour of all items of such a list (e.g., a pointer to the next item, a pure virtual function which returns the size of this object etc.) Having this class, we define a template child of it. The template gets the type of the stored value as its parameter. Each item of such a list would be an instance of the template. Using C++ runtime type identification (RTTI) we can tell one type of an item from another when the list is handled.

Practice shows however this doesn't *completely* satisfy the programmers' needs. Each language grows a special environment where new techniques and methods appear, and these methods usually base on more than one paradigm (such as heterogenous lists). If we remember Lisp working on a C++ project, we probably won't stop with Lisp lists alone. Once we implemented the lists, the next thing we might need is Lisp mapping functions, or Lisp destructive list changing and garbage collection, and so on.

3.8 Summary

Now let's summarize what conditions do we want to meet with a new solution. We need a framework for multiparadigm programming which

1. allows to use one of the languages widely accepted by the industry as it is, i.e. with no changes to the existing compilers and other tools;
2. delivers additional paradigms as they exist in the choosen alternative language, all (or almost all) together;
3. doesn't place any limitations over interparadigm function calls and sharing data between different code;
4. doesn't require lexical and syntactic analyses of any parts of the code at runtime.

In the next chapter we introduce a technique which complies the above requirements. It is discussed assuming C++ is the primary language and Lisp is the language we need to import the set of paradigms from.

4 The key idea

In order to explain the idea of a new technique, let's discuss what do we actually need from the secondary language (e.g., Lisp). Do we, for example, need its syntax? Perhaps we don't. Generally speaking, **we need the paradigms developed around the language, not the language itself**.

Lisp language implements a kind of algebra on so called S-expressions. Both program and data are built of S-expressions. The basic operations on S-expressions are:

- composition (allows to make a list or an arbitrary binary tree of S-expressions)
- decomposition (retriving elements of a list or a tree)
- evaluation (allows to treat an S-expression as a code and perform the according operations)
- lambda (allows to build an S-expression of a functional type, so-called closure, with a given list of formal parameters and a list representing the function's body)

Special type of S-expression called *symbol* (in the terminology tradition to Common Lisp [15]) or *identifier* (in the Scheme's terminology [9]) has additional operations - assigning a value, binding a value and assigning a function. In real dialects of Lisp this set is wider, but we'll limit to these 3 operations.

Besides that, there are additional basic operations (that is, operations which require one to know the internal representation of S-expressions in order to implement such an operation). Some of them are necessary to make the algebra useful (e.g. arythmetic operations on numberic S-expressions), while others are intended for the user's convenience.

The mentioned operations create an algebra on the space of S-expressions. We will denote the introduced algebra as *S-algebra*.

It is clear that S-algebra being implemented in any particular way will give us the full set of Lisp paradigms. S-algebra may be implemented without an actual Lisp interpreter. All we need is to keep it useful, that is, provide a convenient instrumental basis to operate S-expressions and apply all the necessary operations.

In some languages (including C++ and Ada) it is possible to overload standard operations such as `+`, `-`, `/` etc. This allows to implement an arbitrary algebra using very natural and convenient syntax. For example, one can implement a mathematical notion of a vector using `+` for vector addition, `-` for vector difference operation, `*` for the scalar multiplication and (for instance) `^` for vector multiplication.

In the same manner we can implement the notion of S-expression with a class (or, more precisely, with a polymorphic hierarchy of classes) and invent a certain set of operations so as to implement the whole S-algebra presented in Lisp.

5 Representation of S-algebra with C++

In this section the architecture and design of a C++ class library named InteLib [1] is explained as an illustration of the proposed idea.

5.1 S-expressions of various types

The notion of S-expression is represented with an abstract class which for historical reasons is called LTerm. In Lisp, there are S-expressions of different types (numberic constants, string constants, symbols, dotted pairs, functional objects/closures etc.) A polymorphic inheritance technique is used to represent differend types of S-expressions. The LTerm hierarchy is shown at fig. 3.

In the discussed version of the library the `LTerm` class' children serve to represent various types of S-expressions:

- `LTermInteger` and `LTermFloat` represent numberic constants;
- `LTermString` represents string constants;
- `LTermLabel` is introduced to represent S-expressions whose role in the system is determined by the particular instance of the object (such as Common Lisp symbols, as well as `#t` and `#f` in Scheme);
- `LTermSymbol` represents Lisp symbols;

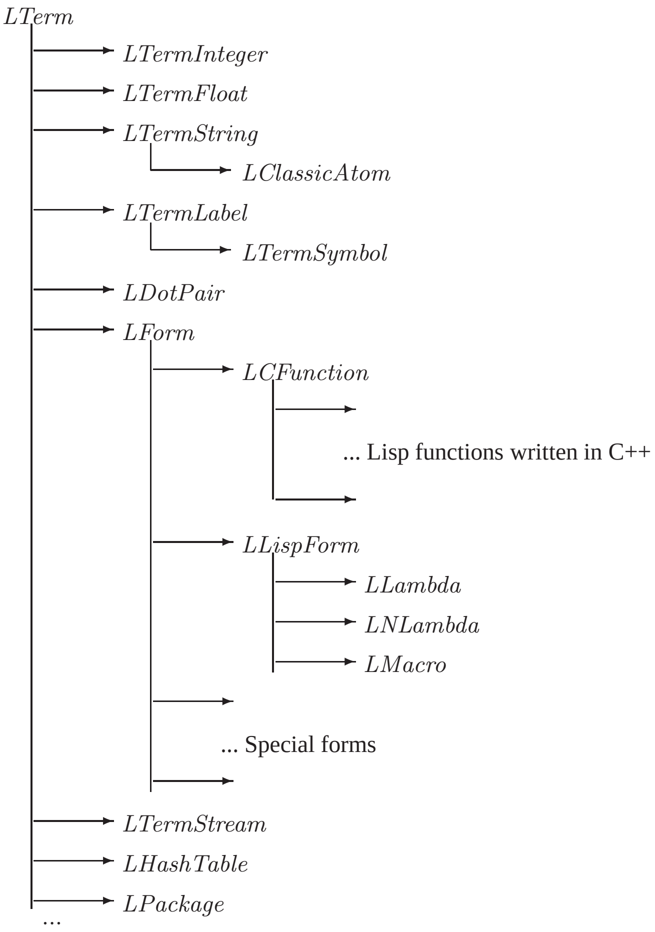

Fig. 3. LTerm classes hierarchy intended to represent S-expressions of various types

- `LDotPair` represents dotted pairs which Lisp lists are built of;
- `LForm` represents generic functional S-expression such as library function, user-defined function or lexical closure, Lisp macro etc.
- several additional classes to represent miscellaneous features such as hash table, i/o stream etc.

InteLib supports two types of numberic constants, namely integers and floats. Compile-time options of the library allows to choose what numberic types we actually need. It is possible to cause `LTermInteger` to use `short`, `int`, `long` or `long long` type to store the actual value, as well as `LTermFloat` can be tuned to use `float`, `double` or `long double` form of a floating point number.

Strings are implemented assuming a string constant itself is never changed. In contrast with Common Lisp, there is no vector type of S-expression in the discussed model so a string is considered as just an atomic value.

There is no special type of S-expression for single characters. They are represented with an `LTermString` object as a string which has length of 1.

Lisp symbols are implemented with class `LTermSymbol`. The class is capable to hold a reference to an object which represents the current dynamic value of the symbol and to another object which represents the function associated with the symbol. Stuff related to lexical bindings of a symbol is implemented outside the class but is supported with its methods (those related to setting and getting the value).

The notion of an empty list may be implemented by any object of S-expression; the only condition is that the address of the object is known at the compile time because the end-of-list check is performed just by comparing pointers. Usually an object of the class `LTermSymbol` (for dialects close to Common Lisp) or LTermLabel (for Scheme-like dialects) are used for this purpose.

To represent functionals as data, `LForm` subhierarchy has been developed within the `LTerm` hierarchy. The subhierarchy has `LCFunction` and `LLispForm` classes derived directly from `LForm`. Lisp special forms are also represented with classes descended directly from `LForm`.

The `LCFunction` class represents functions implemented in C++, including all ”built-in” functions such as `CAR`, `CDR`, `CONS` etc. To add a new Lisp function to the library, a programmer needs to declare a child of `LCFunction` with only one method (`DoCall`) overriden in order to implement the necessary functionality.

The `LLispForm` class is intended to represent forms defined with Lisp constructions (lambda functions, nlambda functions and macros). It has references to the lambda-list (the list of formal parameters), the function body and the lexical context[5] of the form. `LLispForm` class has child classes corresponding to different kinds of forms:

- `LLambda` (ordinary Lisp function which evaluates all its arguments and then evaluates the body in its own context);
- `LMacro` (Lisp macro evaluated as in Common Lisp);
- `LNLambda` (Lisp function which does not evaluate its arguments).

5.2 Garbage collection

Some of `LTerm`’s children are large so that it is inefficient to pass them by value. However, sometimes these objects are constructed within a function which may be called for the value as well as for its side effect so that it is no good to return the created object by pointer (if the function is called for the side effect then the constructed object goes to garbage).

In order to provide garbage collection, another class (called `LReference`) is provided. It has precisely the same size as a pointer do so that it is not so bad to pass it by value. An object of `LReference` class acts just like a pointer to an `LTerm` object

[5] The notion of lexical context is implemented with a separate class LLexicalContext

Table 1. Examples of Lisp expressions representation with C++ constructs

C++ constructs	Lisp equivalent
`(L| 25, 36, 49)`	`(25 36 49)`
`(L| "I am the walrus", 1965)`	`("I am the walrus" 1965)`
`(L| 1, 2, (L| 3, 4), 5, 6)`	`(1 2 (3 4) 5 6)`
`(L| (L| 1, 2), 3, 4)`	`((1 2) 3 4)`
`(L| MEMBER, 1, ˜(L| 1, 3, 5))`	`(member 1 ’(1 3 5))`
`(L| APPEND, ˜(L| 10, 20), ˜(L| 30, 40))`	`(append ’(10 20) ’(30 40))`
`(L| 1 || 2)`	`(1 . 2)`
`((L| 1, 2, 3)|| 4)`	`(1 2 3 . 4)`

having necessary operations including `*` and `->`. `LReference` is intended to be the primary interface to the library. It has a lot of constructors which allow to construct an S-expression from a value of any base C++ data type (integers, floats, strings etc).

In most cases, the objects of `LTerm` class hierarchy reside in dynamic memory and are not operated directly (altough it is possible). `LReference` objects are used to handle `LTerms`.

Besides other features, `LReference` notifies the pointed object when another pointer to it is created or an existing pointer is no longer pointing to it (e.g., it is assigned with another value or destructed). `LTerm` class performs simple reference counting and deletes the object once it has zero references.

Reference counting is choosen for its simplicity. It has well-known problems (including the problem of cyclic constructions). If it is inappropriate for a particular application to use reference counting, then any of existing C++ garbage collection libraries can be used instead. The library has a compile-time option to switch off the reference counting code.

5.3 Lexical bindings

There are also additional classes that represent (in traditional Lisp terminology) a notion of lexical context. Objects of these classes are not operated by user in most cases. The library does not, however, hide them from the user because in some cases it might be useful to create a context manually. There’s always one active lexical context (possibly special null context). In order to use a context it must be activated. Then it is affected by operation of binding a value to a symbol. The context itself affects operations of assignment and retrieving a value of a symbol.

5.4 List composition operations

In Lisp there’s an operation of constructing a list of an arbitrary length denoted by parentheses. The operation has variable ’arity’. For example, a construct `(1 2 3)` has 3 arguments and builds a list of 3 items - 1, 2 and 3. Besides that, there’s a binary operation which builds a dotted pair, such as `(1 . 2)`. The construct `(1 2 3)` has the same effect as a superposition of 3 dotted pair constructors, like this:

```
  (1 2 3)   ==   (1 . (2 . (3 . NIL)))
```

We can implement an operation similar to the Lisp's `( . )` and it will allow us to build any list of S-expressions. It is though a bit inconvenient to create lists using this operation only (imagine you couldn't use plain lists in Lisp, only dotted pairs).

Another problem is that one might want to use just a C++ constant or expression without explicit cast of it to an S-expression, e.g. '3', not a construct like `LReference(3)`, so in case of an overloaded standard operation we need at least one of operands already of LReference type, which allow a C++ compiler to understand we mean the overloaded operation, not a standard one. This requirement also prevents us from using C++ functions with unspecified number of arguments because there's no way to determine at run time what types of actual parameters do we have.

The problem is solved replacing the Lisp '( )' operation of an arbitrary list composition with two operations. First of them creates a list of one element, while second adds an element to a given list. It is clear the two operations allow to create an arbitrary list, that is, their combination has the same functionality as the Lisp '( )' operation.

The first operation always has exactly one argument, but we can't use a symbol of any standard unary operation for it because we want it to be applicable to an expression of any standard type. We also don't want to use a plain function for this purpose because parentheses would make our constructs less clear. The problem is solved with a class `LListConstructor`, which is created to be a label for a binary operation to show the compiler to apply an overloaded one instead of the built-in operation. Usually there's only one instance of `LListConstructor` named `L`. For example, an operator `L|3` returns an `LReference` object that represents Lisp construct `(3)`.

For appending a new item to a list we can overload any overloadable binary operator. For a better clarity we decide to use C++ comma (`,`) for this purpose. Left-hand operand of a comma is always an LReference representing a list. Comma destructively changes the list replacing the final `NIL` with a dotted pair of `(X . NIL)` where `X` is its right-hand operand casted to an `LReference`. This makes it possible to represent Lisp lists in C++ as shown in table 1.

There's a supplementary unary operation in most of Lisp dialects which allows to construct a list of two elements, first of which is a symbol `QUOTE` while the second is the operation's operand. The operation is usually denoted by a single quote symbol (`'`). InteLib overloads the operator ˜ (tilde) for this purpose. See table 1 for an example[6].

For composing dotted pairs and dotted lists InteLib offers an operation `||`. The left-side operand must be a list (possibly of one element). The operand appearing at the right side of the operator is converted to LReference and then the operation replaces the last `NIL` of the given list with whatever it constructed from the right-side operand. See table 1 for an example.

Note the parentheses in the last example. They appear because the comma operator has lower precedence than `||`.

[6] The operator is oveloaded for LReference class so it is possible to apply it to a list or a Lisp symbol, but one can't use it with strings or numberic constants. It is unnecessary anyway to quote them since they always evaluate to themselves

5.5 Operations implemented as regular methods

The most important operation on S-expressions is the *evaluation* of an S-expression. Evaluation is an unary function which maps from $S_x \backslash S_{ux}$ to S_x where S_x is a space of all possible S-expressions and S_{ux} is a set of 'unevaluable' S-expressions (such as closures). Performing evaluation of an S-expression one can also get a side effect.

All constants evaluate to themselves with no side effects. Variables evaluate to their values, if any. Evaluation of an unbound variable generates an error.

Evaluation of a list interpretes the first element as an instruction what functional object to apply to the rest of the list as a list of parameters. In most cases, the resting elements of the list are evaluated and the appropriate function is applied to a list built of the results. The first element of the list must be either a symbol which has an associated functional object or a Lambda-list.

It looks like we need to implement two operations in order to support the evaluation: the evaluation itself (as a polymorphic method of the `LTerm` class) and an operation of *application* of a functional S-expression (a one that belongs to `LForm` subhierarchy) to a list of parameters. The two operations are implemented as methods called '`Evaluate`' and '`Call`', respectively. The '`Call`' method generates an error when called for an object of a non-functional type. '`Evaluate`' generates an error when it is impossible to evaluate the given S-expression. The `IsSelfEvaluated` method allows to determine whether the object represents a constant that always evaluates to itself.

Besides that, there are methods '`Car`' and '`Cdr`' in LTerm class. They return the respective cells of a dotted pair when called for an `LDotPair` object. For an object that represents empty list both methods return empty list. Calling these methods for a non-list object will cause an error.

Another important method named `TextRepresentation` allows to create a human-readable representation of any given S-expression.

It is well known that there are 3 different predicates of equality in Lisp, called `EQ`, `EQL` and `EQUAL`. `EQ` predicate is the simplest one, it just compares two addresses. `EQL` is a bit more flexible. Two objects may be not the same while representing the same value (e.g. two instances of an integer constant 2). In order to make it possible to implement `EQL` predicate for any `LTerm` object there's a virtual method `SpecificEql` which returns false by default. Implementation of `EQL` checks for equality of addresses first, and only if the objects are not `EQ`, it calls `SpecificEql` for one of them passing the other as an argument, so that it doesn't cause a misbehaviour when `SpecificEql` returns false when the compating objects are the same, that is, equal in the sence of `EQ`.

Another important operation, Lambda, is implemented by a constructor of `LLambda` class. For example, expression

```
LReference(new LLambda(NULL, (L| A),
                       (L| (L| PLUS, A, 1))))
```

creates a closure with `NULL` lexical context. The closure takes a numberic S-expression and returns a number which is greater by one. In Lisp such a closure would be represented as `(lambda (a) (plus a 1))`. In order to create a real closure that has lexically bound variables, the appropriate lexical context must be passes as the first argument of the `LLambda` constructor.

There are other methods in classes of the library which a provided mostly for user's convenience. They include typecasting operations, which allow to convert a constant S-expression into a base C++ value. For example,

```
LReference(3)->GetInteger();
```

will return 3, while

```
LReference("Hello world")->GetString();
```

will return a pointer to a constant string `"Hello world"`. Calling such a method for a wrong type of S-expression causes an error.

5.6 Operations performed by standard Lisp functions

Standard, or built-in, functions play a key role in Lisp functionality providing a basis for building programs. They can also be thought as operations on S-expressions, that is, as elements of S-algebra.

In Lisp there are symbols that initially have associated built-in functions. InteLib doesn't provide such symbols in order to allow a user to use whatever names she wants for these symbols. The functional objects representing well-known Lisp functions are direct children of `LCFunction` class (for functions that evaluate all arguments) or of `LForm` class (for special forms). They usually have names such as `LFunctionCar`, `LFunctionCons`, `LFunctionLet`, `LFunctionDefun` and so on.

For convenience there is a generic class

```
template<class F> class LFunctionalSymbol
```

whose argument must be a class that represents a particular function. An instance of `LFunctionalSymbol` differs from `LSymbol` in that its constructor creates an object of the given finctional class and lets it be the associated function of the symbol. For example, one might want to add the following declaration to the program:

```
LFunctionalSymbol<LFunctionCar> CAR("CAR");
LFunctionalSymbol<LFunctionCdr> CDR("CDR");
LFunctionalSymbol<LFunctionCons> CONS("CONS");
LFunctionalSymbol<LFunctionCond> COND("COND");
LFunctionalSymbol<LFunctionDefun> DEFUN("DEFUN");
```

etc. As usual, the constructor's argument sets the textual name of the symbol which is used by `TextRepresentation` method.

Consider, for example, the following Lisp code:

```
(defun isomorphic (tree1 tree2)
 (cond ((atom tree1) (atom tree2))
       ((atom tree2) NIL)
       (t (and (isomorphic (car tree1)
                           (car tree2))
               (isomorphic (cdr tree1)
                           (cdr tree2))))))
```

One can write the module shown at fig. 4 to do the same thing in C++. The module

```
//       File isomorph.cpp
#include "intelib.h"
LSymbol ISOMORPHIC("ISOMORPHIC");
void LispInit_isomorphic() {
 static LSymbol TREE1("TREE1");
 static LSymbol TREE2("TREE2");
 static LFunctionalSymbol<LFunctionDefun> DEFUN("DEFUN");
 static LFunctionalSymbol<LFunctionCond> COND("COND");
 static LFunctionalSymbol<LFunctionAtom> ATOM("ATOM");
 static LFunctionalSymbol<LFunctionAnd> AND("AND");
 static LFunctionalSymbol<LFunctionCar> CAR("CAR");
 static LFunctionalSymbol<LFunctionCdr> CDR("CDR");
 (L|DEFUN, ISOMORPHIC, (L|TREE1, TREE2),
  (L|COND,
   (L|(L|ATOM, TREE1), (L|ATOM, TREE2)),
     (L|(L|ATOM, TREE2), NIL),
     (L|T, (L|AND,
       (L|ISOMORPHIC, (L|CAR, TREE1), (L|CAR, TREE2)),
       (L|ISOMORPHIC, (L|CDR, TREE1), (L|CDR, TREE2)))))).Evaluate();
}
 //      end of file
```

Fig. 4. Example of a C++ module that defines a function in a manner of Lisp

compiles with an ordinary C++ compiler without any additional preprocessing. The symbol `ISOMORPHIC` is public and can therefore be used in other modules.

Please note there are no definitions of symbols `T` and `NIL`. They are provided by the library as well as symbols `QUOTE` and `LAMBDA`. The library needs symbols `T`, `NIL` and `QUOTE` because it is possible to obtain them from certain operations without mentioning them in a program. Consider the following example:

```
     (eql 1 2)   ->  nil
     (eql 1 1)   ->  t
     (car ’’a)   ->  quote
```

The symbol `LAMBDA` can’t be obtained in such a way, but it has special meaning regardless of it’s possible value and/or associated function, so we have to rely on the symbol itself (that is, for example, on the object’s address). That’s why these 4 symbols are provided by the library in contrast with all the other well-known symbols.

It is important to understand that there’s no Lisp as such in the C++ module shown at the fig. 4. The module is written in C++ language. The compiler knows nothing about special meaning of all these commas and vertical bars; they are handled as functions just like in any program which overloads standard operators. Thus we can say we made no changes to the primary language, at the same time allowing a programmer to use paradigms from another (secondary) language. Only paradigms are imported from Lisp, not the language itself. In other words, we **import S-algebra which brings us the paradigms of Lisp without Lisp language as such.**

6 Translation from Lisp to C++

The primary goal of the InteLib library is to bring Lisp paradigms to C++. However, as a side effect it opens a clear way to translation of Lisp code into C++. The translator is made which uses simple rules of transformation of the code. Besides that, the translator generates the necessary definitions of symbols and performs some other tasks as to allow using several Lisp modules within a single project. Since the translator is not the main goal of the project, we don't explain it in details in this paper; we only give a short description.

The translator takes one or more Lisp files and proudces a C++ module (that is, a ”source” file and a header file). The translator understands a certain set of so-called translation directives (top level forms beginning with a token `%%%`), which allow to control how names are translated etc.

The character set for C++ identifiers differs from the traditional one for Lisp symbols. C++ identifiers are case-sensitive and can consist of letters, digits and the underline symbol. The first character of identifier must be a letter or the underline, not a digit. Lisp symbols are case insensitive and may be built of letters, digits and various symols such as `+`, `-`, `*`, `_`, `%` etc. so we need a certain translation algorythm.

The fact that Lisp symbols are case sensitive and C++ identifiers are not is very helpful in translation of names from Lisp to C++. This allows us to bring all letters in a Lisp symbol to the upper case and leave the lower-case letters to represent all these pluses, dashes and other symbols that are not allowed in C++ identifiers. For instance, the Lisp symbol `read-char` might be represented as `READdashCHAR`. If a symbol's name begins from a digit (which is illegal in C++), we prepend it with a lowercase letter, for instance - ”`d`”. The symbol `7seas` having been translated this way becomes the identifier `d7SEAS`.

There are some exceptions from the general rules. For example, the Lisp symbol `null` would be translated to `NULL` producing a name conflict with some of the standard header files. That's why it is translated as `lNULL` (as specified by an appropriate translation directive). User can add her own exceptions, rules etc.

As to experience, using of the translator is good when a whole module is inplemented in Lisp because the traditional Lisp syntax is more convenient than the introduced C++ implementation of S-algebra. It is still possible, however, to avoid using the translator.

The dialect of Lisp recognized by the translator was named InteLib Lisp. It is a very short and simple dialect designed keeping in mind that it is to be used as a secondary language. It ommits many features of modern Lisp dialects because they are considered not to be essential.

7 Logic programming

The explained technique can be applied to secondary languages other than Lisp. One of the obvious ways of further development is to apply it to one of the logic programming languages.

As of now, an attempt is done to apply the technique to a very restrictive subset of Prolog[4]. The Prolog part of the library, unlike the Lisp part, has primarily a demonstration value; creating a library useful in real programming practice is the subject of further work.

The implemented subset of Prolog has no dynamic data structures (lists and functors), just like in Datalog[5]. Unlike Datalog, the implemented dialect has the *cut operation*.

Prolog machines operate on data which is similar to S-expressions (in fact, the only difference is functors). Creating a model of a Prolog machine, a decision was made to reuse the classes already implemented to represent Lisp data[7].

To represent the notion of *predicate*, `DlAbstractPredicate` class is invented. It has a pure abstract method named `DlCreateAbstractIterator` which is intended to create an iterator to fetch, one by one, solutions of a given predicate provided with the appropriate number of arguments.

To create atoms of given predicate (e.g., having predicate `father`, create the atom `father(john, X)`), the class provides `operator()` for 0, 1, 2, ..., 10 arguments of the type LReference[8].

Another class, derived from `DlAbstractPredicate` and named `DlPredicate`, represents the predicate which is the part of Prolog program (that is, a predicate formed of clauses of goals).

Prolog atoms (constructs such as `father(john, mary)`, `vertex(X)`, etc.) are represented with the `DlAtom` class which is in fact just a pair of a predicate and an argument list. Atoms are usually created within functions, locally, so another smart pointer is necessary. It is named `DlAtomRef`. This smart pointer is used as the primary interface to the `DlAtom` class. One of the most important operations of `DlAtomRef` is `operator<<=()` which is used instead of the well-known Prolog symbol `:-`. The operator adds another clause to the appropriate predicate and returns a reference to the object of the class `DlPredicate::DlClause`. That object, in turn, has `operator,()` to add goals (atoms) to it.

Prolog variables are represented using class `DlVariable`.

Using all these classes and operations, we can represent the Prolog clause

```
grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

with C++ expression

```
grandfather(X, Y) <<= father(X, Z), father(Z, Y);
```

[7] So the dialect in fact can handle lists, but it still treats them as atomic data values, e.g., when doing unification

[8] There is no operator with variable parameters list, because LReference objects, being objects of a class, can't be passed through ... in C++.

where X, Y and Z are objects of the class DlVariable. Facts such as

```
        father(john, george).
```

are represented using another form of the operator <<=:

```
        father(john, george) <<= true;
```

Prolog code example:

```
    father(john, george).
    father(george, alex).
    father(alex, alan).
    father(alan, poul).
    grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

Equal C++ code:

```
#include "il_dlog.h"
LSymbol john("john"), george("george"),
  alex("alex"), alan("alan"), poul("poul");
DlVariable X("X"), Y("Y"), Z("Z");
DlPredicate father, grandfather;
void PrologInit_father_grandfather() {
    father(john, george) <<= true;
    father(george, alex) <<= true;
    father(alex, alan) <<= true;
    father(alan, poul) <<= true;
    grandfather(X, Y) <<= father(X, Z), father(Z, Y);
}
```

Fig. 5. Prolog-like C++ code example

Fig. 5 shows an example of a Prolog-like C++ code which uses the predicates father and grandfather. Now, the code

```
      iter = grandfather(X, Y).CreateIterator();
      bool rc;
      do {
        PDlSubstitution solution;
        rc = iter->NextSolution(solution);
        if (rc) {
          printf("%s\n", solution->TextRepresentation().c_str());
        }
      } while (rc);
```

will print the following solutions list:

```
{X/john Y/alex}
{X/george Y/alan}
{X/alex Y/poul}
```

As we already noted before, the implemented model doesn't do unification of dynamic data structures though Lisp lists can be operated with, considering them atomic datums. To produce a more interesting demo, let's add a built-in predicate named `DLCONS(car, cdr, cons)`. The predicate is implemented by another class derived from `DlAbstractPredicate`, named `DlPredicateCons`. The predicate is able to work having any of its arguments specified or unspecified (that is, a variable is given instead of a value).

Another useful predicate is `DlPredicateLispcall` (to call the Lisp machine explained before).

The object `DlCut` is a special value of `DlAtomRef` which represents the cut operator.

Note also that a `DlPredicate` without any clauses always fails, so to implement an *always failing* goal we can just create an empty `DlPredicate`.

Having all these objects, we are ready to write a simple program which finds a path in a given graph (fig. 6).

Now, if we create the appropriate iterator with

```
iter = Shortpath(2, 4, X, 2).CreateIterator();
```

the solution finding code like the one shown above will print the solution

```
{X/(2 1 4)}
```

Unlike the Lisp part of InteLib which is already useful in some practical cases, the explained Prolog part is only a simple demo. It is planned to implement a more practically useful library in the close future.

8 Conclusions

The most important advantage of the proposed technique is that there's no need for two programming systems within a project. The existing C++ compiler is always used, and the only thing required to use the technique is a C++ class library which has a relatively simple imterface.

It is also possible to use another primary language. The only requirement to it is the possibility of overloading of standard operations. In particular, Ada95 may be used as the primary language as well (at least for modelling Lisp as the secondary language). Implementation of the appropriate library for Ada95 might be one of the further work goals.

```
LListConstructor L;
DlPredicateCons DLCONS;
DlPredicateLispcall DLLISPCALL;
DlAbstractPredicateIterator *iter;
DlVariable X("X");
DlVariable Y("Y");
DlVariable Z("Z");
DlVariable P("P");
DlVariable N("N");
DlVariable V1("V1");
DlVariable V2("V2");
DlVariable V3("V3");
DlPredicate Edge("Edge");
DlPredicate Edge2("Edge2");
DlPredicate Member("Member");
DlPredicate Shortpath("Shortpath");
DlPredicate Fail("Fail");
Member(X, Y) <<= DLCONS(X, V2, Y);
Member(X, Y) <<= DLCONS(V1, V2, Y), Member(X, V2);
Edge(1, 2) <<= true;
Edge(1, 3) <<= true;
Edge(1, 4) <<= true;
Edge(1, 5) <<= true;
Edge(5, 3) <<= true;
Edge2(X, Y) <<= Edge(X, Y);
Edge2(X, Y) <<= Edge(Y, X);
Shortpath(X, Y, P, 1) <<=
             DlCut,
             Edge2(X, Y),
             DLCONS(Y, L, V1),
             DLCONS(X, V1, P);
Shortpath(X, Y, P, N) <<=
             DLLISPCALL((L|lt, N, 1), T),
             DlCut,
             Fail();
Shortpath(X, Y, P, N) <<=
             DLLISPCALL((L|minus, N, 1), V1),
             Shortpath(Z, Y, V2, V1),
             Edge2(X, Z),
             DLCONS(X, V2, P);
```

Fig. 6. Graph path finding program

Bibliography

[1] E. Bolshakova and A. Stolyarov. Building functional techniques into an object-oriented system. In *Knowledge-Based Software Engineering. Proceedings of the 4th JCKBSE*, volume 62 of *Frontiers in Artificial Intelligence and Applications*, pages 101–106, Brno, Czech Republic, September 2000. IOS Press, Amsterdam.

[2] G. Booch. *Object-oriented Analyses and Design*. Addison-Wesley, Reading, Massachusets, second edition, 1994.

[3] T. A. Budd. *Multy-Paradigm Programming in LEDA*. Addison-Wesley, Reading, Massachusets, 1995.

[4] A. Calmerauer, H. Kanoui, and M. van Caneghem. Prolog, bases théoriques et développements actuels. *Technique et Science Informatiques*, 2(4):271–311, 1983.

[5] S. Ceri, G. Gottlob, and L. Tanka. *Logic Programming and Databases*. Springer-Verlag, Berlin, 1990.

[6] U. W. Eisenecker. Future trends in multi-paradigm programming. Position Paper for the ECOOP'98 Panel on Multi-Paradigm Programming, 1998.

[7] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, Reading, Massachusets, 1998.

[8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusets, 1983.

[9] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on Algorithmic Language Scheme, 1998.

[10] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[11] M. Müller, T. Müller, and P. Van Roy. Multiparadigm programming in Oz. In D. Smith, O. Ridoux, and P. Van Roy, editors, *Workshop on the Future of Logic Programming*. International Logic Programming Symposium, 1995.

[12] J. Robinson. Logic programming - past, present and future. *New Generation Computing*, 1:107–121, 1983.

[13] D. Spinellis, S. Drossoupoulou, and S. Eisenbach. Language and architecture paradigms as object classes: A unified approach towards multiparadigm programming. In J. Gutknecht, editor, *Programming Languages and System Architectures International Conference*, volume 782 of *Lecture Notes in Computer Science*, pages 191–207, Zurich, Switzerland, March 1994. Springer-Verlag.

[14] D. D. Spinellis. *Programming paradigms as object classes: a structuring mechanism for multiparadigm programming*. PhD thesis, University of London, London SW7 2BZ, United Kingdom, February 1994.

[15] G. L. Steele. *Common Lisp the Language*. Digital Press, second edition, 1990.

[16] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusets, third edition, 1997.

[17] V. Turchin. *REFAL-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, 1989.

[18] D. A. Turner. Miranda – a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Proceedings of the Conference of Functional Program-*

ming Languages and Computer Architecture, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, 1985. Springer-Verlag.

Program Templates:

Expression Templates Applied to Program Evaluation

Francis Maes

EPITA Research and Development Laboratory,
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France,
francis.maes@lrde.epita.fr,
WWW home page: http://lrde.epita.fr/

Abstract. The C++ language provides a two-layer execution model: static execution of meta-programs and dynamic execution of resulting programs. The Expression Templates technique takes advantage of this dual execution model through the construction of C++ types expressing simple arithmetic formulas. Our intent is to extend this technique to a whole programming language. The Tiger language is a small, imperative language with types, variables, arrays, records, flow control structures and nested functions. The first step is to show how to express a Tiger program as a C++ type. The second step concerns operational analysis which is done through the use of meta-programs. Finally an implementation of our Tiger evaluator is proposed.

Our technique goes much deeper than the Expression Templates one. It shows how the generative power of C++ meta-programming can be used in order to compile abstract syntax trees of a fully featured programming language.

1 Introduction

During the compilation process, an input program expressed in textual form is transformed by successive steps into executable code. As in any language, a C++ program will basically be evaluated during its execution. The interesting particularity of C++ is its ability to do some computations at compile-time using template constructions (the so-called meta-programs, see [12], [3], [7] and appendix A for an example). This two-layer execution model corresponds to the usual concept of static (compile-time) and dynamic (execution-time) processing.

In C++, there is a technique called Expression Templates described by [11], which allows the exploitation of this two-layer execution model. This technique relies on transformations of simple arithmetic expressions at compile-time to increase the performances of the executable code. Moreover some evaluation can be done entirely statically with mechanisms such as constant propagation. This way, some computations usually done at execution-time are processed at compile-time.

The Expression Templates technique is based on the use of template classes. In order to work on expressions, we need a structural description of them. This is done by building a type that reflects the abstract syntax tree (AST) of the expression. Each node of this tree will be translated into a template class whose arguments are the node subtrees.

Usually, a program written in any language can also be expressed as an abstract syntax tree. The next natural step is to wonder whether it is possible to extend the Expression Templates technique to a whole programming language. Expressing a full program with a C++ type reflecting its AST could thus be made possible. In the remainder of this paper, this type will be called the TAT (Tree As Type). A TAT is a representation of an AST using a C++ type formalism.

Expressing a program in the TAT formalism would allow us to adapt the Expression Templates evaluation method to a whole program and therefore to take advantage of the two-layer execution model of C++ (see [5]). The entire process of compiling and executing a program expressed as a TAT corresponds to its evaluation.

To experiment this idea, we have to choose a programming language that does not have this two-layer execution model. We want this language to be simple and to have few constructions. Nevertheless, this language must at least include types, functions, records, arrays and flow control constructions. Tiger, a language defined by [1], corresponds to our needs: with only 40 rules in its EBNF grammar, it respects all our conditions.

This work is a proof of concept. No-one had previously mapped an entire language to a C++ meta-program. Those that consider C++ expression templates for prototype implementations should be interested in this project. Moreover, the C++ metalanguage is here introduced as an intermediate language. This point of view is different from the current trend of supporting meta-programming by designing metalanguages as extensions of existing programming languages. Our work initially inspired by Expression Templates goes very deeply into the possibilities of C++ meta-programs using several techniques discovered recently.

This paper begins with an overview of related work. Next, section 3 introduces the Tiger language, followed by a description of our architecture. Our first objective is to translate Tiger programs into TATs. When trying to do this, several problems arise (e.g. expressing lists). These are developed in section 4. Our second objective is to do some static processing on this TAT. This will require a structure called environment, and a form of static pointers detailed in section 5. Finally we want to evaluate a Tiger program expressed as a TAT using the C++ two-layer execution model. The implementation which allows this is described in section 6. This is followed by some interesting results related to this new technique. This paper will finish with a discussion about the possibilities of such mechanisms.

2 Related work

Our work is based on Expression Template. The Expression Template is at the basis of our work. This technique described by [11] has many known interests. In particular it allows to build the static AST of a C++ expression. This allows C++ meta-programs to work on C++ expressions seen as types. This can be useful for:

- **Rewriting statements into equivalent (but more efficient) ones**. This was the original intent of Expression Templates. This technique was first used to evaluate vector and matrix expressions in a single pass, without temporaries.

- **Building lambda terms**. Several libraries for doing functionnal programming in C++ are based on Expression Templates. Thanks to C++ meta-programs, several functionnal operations are possible on these lambda terms. The Fact library ([9]) provides typical functional features such as currying, lambda expressions and lazy evaluation in C++. The Boost package also includes a library specialized in lambda expressions: the Boost Lambda Library ([6]). FC++ ([8]) is a similar library inspired by the Haskell language. Our work has something to do with lambda term manipulations: we also manipulate TATs. But our intent is not to do functionnal operations on a TAT but to compile a whole program including functions and variables declarations.
- **Building any other structured expressions**, such as the [4] library which uses Expression Template in order to build EBNF rules. C++ meta-programs are then used to transform a grammar into a usable parser. In this library, C++ meta-programs deal with complex operations such as in our work.

The Expression Template is very useful but a bit complex to implement. PETE ([2]) is a tool that aims at generating the needed code. Fact is built on top of PETE. This tool could help us to build a C++ front-end to our compiler. The idea of using template constructions in compilers has already been used for building a java compiler, see [10].

3 Tiger evaluation and compilation

3.1 Tiger constructions

Tiger is an Algol-style language with a functional flavor. Two kinds of construction exist: declarations and typed expressions. Declarations are of three kinds: type, variable and function declarations. Four basic types exist: integers, strings, nil and void. New types can be built with records and arrays. Existing types can be renamed by a typedef mechanism. Tiger is not a first-order functionnal language: functions cannot be passed as parameters, neither as results.

Tiger has a nested `let-in-end` construction which makes it possible to declare nested scopes. A particular case of this is the ability to declare nested functions.

Except declarations, everything in Tiger is an expression: literals (strings and integers), unary and binary operations, left-values, function calls, array and record instantiations and flow control constructions: `if-then-else`, `while-do`, `for-to-do`, `break`.

3.2 Architecture

We use a front-end program which parses Tiger and does the semantic analysis: type checking, scopes and bindings. The output of this front-end is a C++ program which declares a TAT. Our front-end is based on techniques explained by [1].

The interesting thing is the remaining work: the program evaluation. This task is done in C++ through the static and the dynamic processing.

Our front-end associated with the C++ static processor is a compilation chain. Indeed the input of this chain is a textual Tiger program, and its output is an executable program.

3.3 Comparison with a standard compiler

A usual object oriented compiler first parses the program. It provides AST classes that are dynamically instantiated in order to build the programs abstract tree. At this point until the end of the compilation, successive transformations are applied until getting the executable code.

In our case, we provide a set of template classes corresponding to each node of the AST. During the compilation of a Tiger program, these templates are filled by our front-end giving us the TAT. At this point, the C++ compiler does successive transformations until getting the executable code.

An analogy can easily be done between our Tiger compiler and a standard compiler. Where a standard compiler provides AST classes, we provide AST meta-classes. Where a standard compiler builds an AST expressed as objects, we build an AST expressed as a type (the TAT). A standard compiler provides classes for operational analysis, we provide meta-classes to do this work.

It has been shown that a Turing machine could be constructed with template constructs ([12]). Any work traditionally done by a standard compiler can theoretically be done with C++ meta-programs. The method that we present should thus be adaptable to any other language. The only restrictions are the C++ compilation times and memory use.

4 Translation into TAT

Let us return to the Expression Templates technique with the following Tiger program:

```
 (5 * 10 + 1)
```

Since the Expression Templates technique was originally used to describe and evaluate simple expressions (literals, variables, unary, binary and potentially n-ary operations), such examples can easily be constructed with it. Here is an example of TAT corresponding to the previous example:

Listing 11.1. A simple TAT

```
typedef BinOp< BinOp< ConstInt<5>, ConstInt<10>, Times >,
                                ConstInt<1>, Plus >
                program_t;
```

However this covers a very small part of the whole programming language. Important features such as type declarations, function declarations and calls, or flow control cannot be expressed. Moreover, Tiger expressions are typed: we want our compiler to be able to evaluate and work on typed-expressions. When trying to translate more complex examples into TATs, different problems arise such as the list problem, or the reference problem.

4.1 The list problem

Let us consider this Tiger example:

Listing 11.2. Two functions

```
let
  function double(x : int) : int = 2 * x
  function sum(a : int, b : int, c : int): int = a + b + c
in
  double(30) - sum(6, 1, 2)
end
```

When building this program's TAT, we need to express lists: declaration lists, function formals lists, and function call arguments lists. The usual way to do this is to use recursive lists. A recursive list is defined as empty or as a head element followed by a tail list.

This can be transposed into C++ with the static list technique described by [12]. We use a template class List, which parameters are the first element (a type), and the remaining list. A class EmptyList is used to mark the end of the list. With this notation, we can express lists as types. For example, in `sum (6, 1, 2)`, the argument list can be expressed with the following TAT:

```
List< ConstInt<6>,
  List< ConstInt<1>,
    List< ConstInt<2>,
      EmptyList
        >
      >
    >
```

The full TAT conversion of a similar sample is given in the next section. Static lists, which are a particular case of trees, will be used extensively in the remaining of this paper: this is our first addition to the Expression Templates technique.

4.2 The reference problem

The following simple example illustrate the reference problem:

Listing 11.3. Two variables addition

```
let
  var i : int := 80
  var j : int := 6
in
  i + j
end
```

The expression `i` refers to the variable declaration `var i : int := 80`. The same way, the declaration `var i : int := 80` refers to the builtin type `int`. This example demonstrates that we cannot consider programs as simple trees. The main structure acts as a tree, but the implicit relations by reference transforms this tree into a DAG (direct acyclic graph).

The TAT has to describe a tree plus some graph relations between a declaration and its uses. This is the main difficulty compared to the Expression Templates technique. Without a reference mechanism, we cannot express concepts such as types or functions.

Each time a declaration is referred, we need a pointer to it. The following part shows how to solve this: each declaration will have a location in an evaluation environment.

5 Evaluation Environment

At every point in the program, there is a set of active declarations which can be used. An expression such as `i + j` (listing 11.3), or `double(30) - sum(6, 1, 2)` (listing 11.2) cannot be evaluated without the declaration context: we need to maintain an environment at evaluation time.

Tiger defines some builtin types and functions. These declarations, visible at every point in every Tiger program, will be the initial state of our environment. Declarations that have the same visibility are grouped into scopes. In the remainder of this paper, the list of declarations of the same scope is called a *chunk*.

The main operations we need on this environment are pushing and popping chunks. Moreover, we need a way to extract a declaration, given its chunk and its location in the chunk.

New declarations are introduced with the `let-in-end` structure, which is composed of two parts. A first declarative part, located between `let` and `in`, allows declaring a chunk. The second part, is an expression, in which we can use previous declarations. Evaluating the whole structure is done by pushing the chunk into environment, evaluating the expression and finally popping the chunk.

The environment can also be modified by a function call: when this occurs the evaluation point is changed. This implies that the set of active declarations changes.

Listing 11.4. A function call

```
let
  function double(x : int) : int = 2 * x
in
  let
    var i : int := 17
  in
    double(i) + i
  end
end
```

In the above example, the function call is evaluated the following way:

1. Evaluate function parameters: here i = 17.
2. Initialize formal values: x ← i
3. Pop declarations introduced between the function declaration and the function call: this restores the environment of the function implementation. In our case: pop the chunk containing `var i : int := 17`, as the function double does not know this declaration.

4. Push formals declarations. Here: push a chunk containing `x : int`.
5. Evaluate the function body: `(2 * x)`
6. Restore callers environment: `x` does not exist any more, `i` is reintroduced.

At this point, a stack seems to be appropriate for our needs. This stack will be filled with declaration chunks. A declaration chunk simply contains the corresponding part of the TAT. At a given evaluation point, each visible declaration is located with a pair of indexes: the index of the chunk, and the index of the declaration in the chunk. So a simple pair of indexes is enough to refer to a declaration.

The example 11.3 can now be translated into the following TAT:

```
LetInEnd<
  List< Var< ConstInt< 80 >, builtin_types , int_type >,
    List< Var< ConstInt< 6 >, builtin_types , int_type >,
      EmptyList > >,
  BinOp< SimpleVar< 0, 0 >, SimpleVar< 0, 1 >, Plus >
>
```

The pair $< 0, 0 >$ refers to the first declaration of the first chunk, which corresponds to `var i:= 80`. The pair $< 0, 1 >$ refers to `var j:= 6`. `builtin_types` and `int_type` are predefined integer values, which identify the builtin `int` Tiger type. This mechanism of environment and location pair is a form of static pointers.

We are also able to translate example 11.4:

```
LetInEnd<                                     let
  List< Function< List<                         function double(
      TypeLnk< builtin_types , 1 > >,                   x : int) =
    BinOp< ConstInt< 2 >,                           2
      SimpleVar <1, 0 >, Times >,                     * x
     0 > >,
    LetInEnd<                                 in
      List< Variable< ConstInt< 17 >,           let
                 builtin_types , 1 > >,           var i : int := 17
                                                in
        BinOp< FuncCall< 0, 0,                    double(
            List< SimpleVar< 1, 0 > > >,                 i)
        SimpleVar< 1, 0 >, Plus >                    + i
     >                                          end
>                                             end
```

Let's remember the goal: translating an AST into a C++ type (the TAT), so that the compiler can work on this type. In the proposed implementation, the environment related computations are done at compile-time. Meta-programming techniques will allow us to reduce the execution-time work considerably.

6 Implementation

The basis of the Expression Templates technique is to write a template class per kind of node available in the AST. The parameters of this template are the node subtrees. Each of these template classes correspond to a node of the AST.

These template classes fulfill two roles: first they express the AST information. This is implicitly done with class organization into the TAT. Second, our classes must provide evaluation code.

In the case of expressions, this consists on two tasks: the type calculation, and the value calculation. The declaration classes provide some other services such as common operations for types.

Apart from AST meta-classes, we also need to provide meta-code to perform some static processing. This corresponds to the set of operations related to the evaluation environment.

6.1 Global organization

Two kinds of classes have to be written: expression classes and declaration classes. Declarations will be further distinguished via classes specialized for type, variable and function declarations. Moreover, the implementation also includes the environment mechanism, and tools for its manipulation.

Note that the base classes `AstNode`, `Expression`, `Declaration`, `TypeDec`... are only used for some static checking. These classes are not very interesting, and will not be detailed in this paper.

6.2 Expression classes

As in the Expression Templates technique, each Expression class will implement an evaluation method. These methods are inlined, so that the C++ compiler can build efficient evaluation code.

The main difference with Expression Templates is due to the evaluation environment: The evaluation method depends on the current environment. Another striking difference is that expressions are typed. Evaluating an expression consists in computing both its type and value. We want expression types to be evaluated statically: this work will be done through typedefs. All the typed values that we manipulate are represented with four bytes. In order to simplify, we decided to represent all variables with the `void*` type. This lead us to the following model adopted by all expression classes:

```
// var_t represent a non-typed value.
typedef void* var_t;

// Here comes the template parameters: the TAT subtrees.
template< ... >
struct AnExpression: public Expression
{
  // Evaluation is dependent of current environment.
  template<class T_env>
  struct eval
  {
    // statically compute the expression type
    typedef ... T;
    // inline method that evaluates the expression value
    inline var_t doit ()    { ... }
  };
};
```

```
template<signed Value>
struct ConstInt: public Expression
{
  template<class T_env>
  struct eval
  {
    typedef IntType         T;
    inline var_t doit ()    {return (var_t)Value;}
  };
};
```

ConstInt $< 123 >$ is a TAT: its value and type can be evaluated:

```
  typedef ConstInt<123>   program_t;

  var_t value = program_t::eval< initial_env_t >::doit();
  typedef program_t::eval< initial_env_t >::T  type;
```

Notice that "`::`" is C++ for Java "`.`"

Two types are predefined:

- `var_t` represents all Tiger variables. For example, an `int` can directly be casted into a `var_t` (these two types have the same size: four bytes). Most of time a `var_t` corresponds to a record pointer or an array pointer.
- `initial_env_t` corresponds to the Tiger builtin environment: builtin types such as `IntType` or `StringType`, and builtin functions (`print`, `ord`, `concat`...).

The TAT given in listing 11.1 can now be evaluated. Here is the template expansion chain that leaded to the result: 51.

```
1. program_t::eval< initial_env_t >::doit()
2. BinOP< ConstInt<5>, ConstInt<10>, Times >
                 ::eval< initial_env_t >::doit() +
     ConstInt<1>::eval< initial_env_t >::doit();
3. ConstInt<5>::eval< initial_env_t >::doit() *
   ConstInt<10>::eval< initial_env_t >::doit() + 1;
4. 5 * 10 + 1
5. 51
```

6.3 Declaration classes

The first role of declaration classes is to store information relative to the declaration. For example a variable declaration must store its type and initial value. This is done with template parameters exactly as above. The second role of declaration classes depends on the kind of declaration. For variables and functions, only some utility functions are implemented. The type classes do more things: their second role is to implement all operations related to the type: assignment, comparison, creation and destruction. These operations can depend on the environment. This is for example the case for an array, which refers to the type of its elements.

Here is the model of type declaration classes:

```
struct AType: public TypeDec
{
  // Type eval depends on environment
  template<class T_env>
  struct eval
  {
    // Common operations are implemented here
    void create(var_t& v);
    void destroy(var_t v);
    void assign(var_t& left, var_t right);
    int compare(var_t left, var_t right);
  };
};
```

Such classes are implemented for `VoidType`, `IntType`, `StringType`, `ArrayType` and `RecordType`.

Each new type definition in a Tiger program, will result in new type operations. Our Tiger compiler generates evaluation code, but also operations code. In order to emphasis on this contribution, we chose to implement assignment and comparison as structural. At the contrary to the Tiger specifications, when two records are compared, this is done member by member. When an array is assigned, the all content is copied.

6.4 Program Environment

We have seen that type and expression evaluations depend on an environment, through the type identified by `T_env` in the previous code samples. We want the environment to be computed statically: we need an implementation which allows to push, pop, and retrieve declarations at compile-time. Therefore we use again static lists: an environment is implemented as a static list of declaration chunks. A declaration chunk is a part the TAT which is also a static list. This construction allows us to manipulate the environment:

Pushing and popping declaration chunks is done with typedefs:

```
// Push T_new_chunk on T_env, yielding T_new_env.
typedef List< T_new_chunk, T_env >        T_new_env;

// Pop an element of T_env, yielding T_new_env.
typedef T_env::tail                       T_new_env;
```

Environment access is done with a template class and a specialization:

```
template<class T_env, unsigned N>
```

```
struct ListGet
{
  typedef ListGet<T_env::tail, N - 1>::T  T;
};

template<class T_env>
struct ListGet<T_env, 0>
{
  typedef T_env::head      T;
};

// Access to the chunk number 3.
typedef typename ListGet<T_env, 3>::T           T_chunk_3;
// Access to the declaration number 1 of this chunk.
typedef typename ListGet<T_chunk_3, 1>::T       T_decl_3_1;
```

We are now able to write a simplified version of the `LetInEnd` template class.

```
template<class T_decl, class T_exp>
struct LetInEnd
{
  template<class T_env>
  struct eval
  {
    typedef List<T_decl, T_env>             T_new_env;
    typedef T_exp::eval<T_new_env>::T       T;
    var_t doit()
    {
      // Create new variables declared in T_decl
      // (not detailed here).

      // Evaluate the expression in the new environment.
      var_t res = T_exp::eval<T_new_env>::doit();

      // Destroy the variables declared in T_decl
      // (not detailed here).

      return res;
    }
  };
};
```

All the needed operations on the environment can be done with type operations: we are able to fully compute the environment at compile-time for each evaluation point. Function calls are not detailed here, but they use the same operations. Note that all functions are evaluated each time they are called (as inline functions). This implies that if we want a recursive function to be translated as a C++ recursive function, we need the environment to be exactly the same at each recursive call.

6.5 The dynamic part

Not everything can be done at compile-time. The Tiger language allows some constructions which cannot be resolved statically.

The main dynamic stuff is the variable declaration and use. When a variable is declared, we need to store its value somewhere. At each evaluation point of the program, there is a set of variables which are accessible.

A variable can be of any supported Tiger type: it can be an array, a string or a record. There is no static representation of such values: we need to store this into memory at the program execution. Therefore we use the C++ stack: variables declared in a `let-in-end` construction are declared as local variables in the `LetInEnd` evaluation method.

The Tiger has a nested let declaration. At a given evaluation point, there can be several visible scopes. This obliges us to maintain a stack of scope pointers during the whole execution process. Accessing a variable is performed with two indirections: a first one to get the right scope and another one to reach the variable into this scope. We could have chose to implement variables access with a static link mechanism. This would corresponded to the adaptable closure present in the phoenix library, part of the spirit project ([4]).

These indirections are our main limitation to really perform a static resolution of programs. Conversely, here is a program that is *entirely* statically evaluated:

Listing 11.5. A program solved statically

```
let
  function foo() = 20 * 20
  function bar() = 30 / 2
  function smousse() = if (80 > 6) then 1 else 0
in
   (foo() + bar() + smousse()) * 4
end
```

There is no variable used so, after our transform towards C++, we can expect that a C++ compiler can statically solve this program. In this particular case, using a C++ compiler which has good optimization capacities, we directly obtain one assembler instruction which gives the integer result.

6.6 The C++ program

The C++ program always have the same structure:

```
// Include all template classes needed to express and evaluate
  the AST.
#include "all.h"

// Generate the TAT.
typedef ... program_t;

int main()
{
  // instantiate program evaluation
  return (int)program_t::eval< initial_env_t >::doit();
}
```

The line of the main() launch the doit() instantiation, which results in the generation of the program evaluation code. This work is done by the C++ compiler.

7 Results

Our compiler covers all of the Tiger language. Lot of Tiger programs have been tested, and work successfully. Our process has been tested with como, g++ 3.2 and icc which gives slightly faster programs.

To experiment the performance of generated code, some Tiger programs compiled with our process have been compared to their C hard-coded equivalent. In average, the C program goes two to three times faster than the (C++) Tiger one. This performance lack is mostly explained by the variable access cost: each access needs two indirections. But viewed as an evaluation process, this can be considered as good results.

This performance highly depends on the aptitude of the C++ compiler to optimize code. These optimizations are essentially obtained by the inlining mechanism. This optimization has been tested using the g++ option called `-finline-limit`. This option influences the quantity of functions inlined. This experience showed us the importance of good inlining at compile-time. Optimizations are done until approximatively `-finline-limit-1000`, which is much more than for usual C++ programs. This can be explained by the amount of functions that are instantiated. Indeed for each node of the AST, there is at least one function which will be used.

8 Conclusion

We have seen that a program can be expressed as an Abstract Syntax Tree (AST) given the language grammar. Using a technique based on Expression Templates, we are able to build a C++ type which describes this AST. This representation is called the TAT (Tree As Type).

Building and evaluating the TAT poses various problems. We need to express lists (for declarations, arguments, etc.). This problem is solved using the Static list technique. In the TAT, some elements refer to others. The reference problem implies the use of an environment which is implemented using a stack. We have seen that this container allows the required operations: pushing, popping and accessing. This stack is directly filled with parts of the TAT: this is a form of static pointers, which solves the reference problem.

An implementation based on the Tiger language has been proposed. This implementation intensively uses meta-programming techniques, therefore, the C++ compiler is able to do lot of work at compile-time: expression types and element references are solved statically. The limits of static resolution is the use of variables which can only be manipulated dynamically.

Our Tiger compiler is originally inspired by the Expression Templates technique. However, the evaluated constructions are not restricted to basic ones, such as unary or binary operators, but includes the common flow control constructions, structured types, variables, and nested functions. Moreover, thanks to the use of a static environment, such advanced operations can be evaluated by jumping from one point of the program to another. This happens for example each time a function is called. That characteristic is a noticeable difference with the Expression Templates which are evaluated in a simple bottom-up fashion.

This original technique shows how we used C++ meta-programming in order to work on abstract syntax trees of a mostly functional programming language. Indeed the C++ generative power allowed us to implement compiler parts and translation into C++ equivalent code.

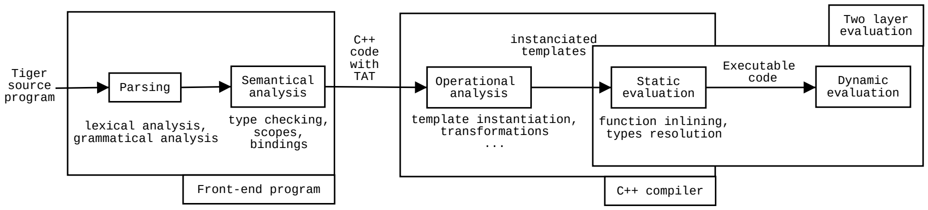

Fig. 1. Placement in the compilation chain

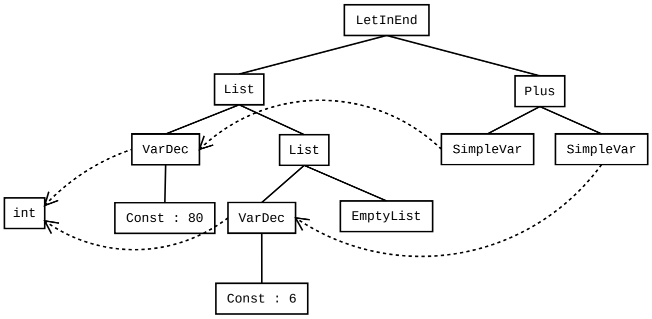

Fig. 2. AST of example 11.3

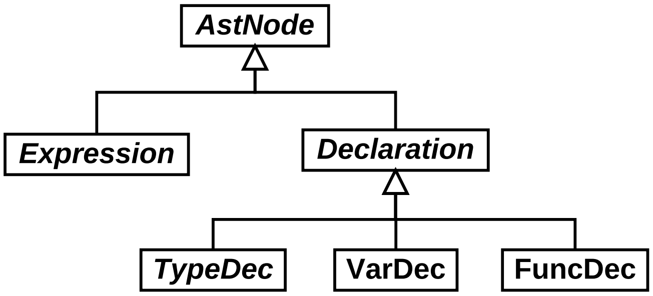

Fig. 3. Main kind of classes

Bibliography

[1] A.W. Appel. *Modern Compiler Implementation in C / Java / ML*. Cambridge University Press, 1997.

[2] J.A. Crotinger, J. Cummings, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith, and T.J. Williams. Generic programming in POOMA and PETE. In *Generic Programming, Proceedings of the International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 218–. Springer-Verlag, 2000.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 2000.

[4] Spirit group. Spirit parser framework, 2002.

[5] S. Haney and J. Crotinger. How templates enable high-performance scientific computing in C++. *Computing in Science and Engineering*, 1(4), 1999.

[6] G. Powell J. Jarvi. The boost lambda library, 2002.

[7] J. Järvi. Compile time recursive objects in C++. In *Technology of Object-Oriented Languages and Systems*, pages 66–77. IEEE Computer Society Press, 1998.

[8] Brian McNamara and Yannis Smaragdakis. Functional programming in C++ using the FC++ library. *SIGPLAN Notices*, April 2001.

[9] Jörg Striegnitz and Stephen A. Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[10] C. van Reeuwijk. Rapid and robust compiler construction using template-based metacompilation. In *12th International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 247–, Warsaw, Poland, April 2003. Springer-Verlag.

[11] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[12] T. Veldhuizen. Techniques for scientific C++. Technical report, Computer Science Department, Indiana University, Bloomington, USA, 2002.

A A simple C++ meta-program and its evaluation

```
template<unsigned i>
struct factorial
{
  enum {res = i * factorial< i - 1 >::res};
};

template<>
struct factorial<0>
{
  enum {res = 1};
};

enum { fact4 = factorial<4>::res };
```

Thanks to the template expansion mechanism, this C++ meta-function allows to compute a factorial at compile-time:

```
factorial<4>::res
4 * factorial<3>::res
4 * 3 * factorial<2>::res
4 * 3 * 2 * factorial<1>::res
4 * 3 * 2 * 1 * factorial<0>::res
4 * 3 * 2 * 1 * 1
24
```

B A full tiger program

```
let
 type any = {any : int}
 var buffer := getchar()

 function printint(i: int) =
  let function f(i:int) = if i>0
             then (f(i/10); print(chr(i-i/10*10+ord("0"))))
   in if i<0 then (print("-"); f(-i))
      else if i>0 then f(i)
      else print("0")
  end


function readint(any: any) : int =
 let var i := 0
     function isdigit(s : string) : int =
                  ord(buffer)>=ord("0") & ord(buffer)<=ord("9")
     function skipto() =
       while buffer=" " | buffer="\n"
         do buffer := getchar()
  in skipto();
     any.any := isdigit(buffer);
     while isdigit(buffer)
       do (i := i*10+ord(buffer)-ord("0"); buffer := getchar());
     i
 end

 type list = {first: int, rest: list}

 function readlist() : list =
    let var any := any{any=0}
```

```
        var i := readint(any)
     in if any.any
         then list{first=i,rest=readlist()}
         else nil
    end

 function merge(a: list, b: list) : list =
   if a=nil then b
   else if b=nil then a
   else if a.first < b.first
      then list{first=a.first,rest=merge(a.rest,b)}
      else list{first=b.first,rest=merge(a,b.rest)}

 function printlist(l: list) =
   if l=nil then print(”\n”)
   else (printint(l.first); print(” ”); printlist(l.rest))

   var list1 := readlist()
   var list2 := (buffer:=getchar(); readlist())

 in
        print (”list 1 : \n”);
        printlist (list1);
        print (”list 2 : \n”);
        printlist (list2);
        print (”merged list : \n”);
        printlist (merge (list1, list2) )
end
```

C TAT of the previous program

The following program compiles in less than two minutes with g++ 3.2 on a 350Mhz processor.

```
#include "all.h"

typedef LetInEnd< List< RecordType< List< TypeLnk< builtin_types, 1 > > > >,
LetInEnd< List< Variable< FuncCall< builtin_funcs, 9, List< > >, builtin_types, 2
    > >,
LetInEnd< List<
Function< List< TypeLnk< builtin_types, 1 > >, LetInEnd< List<
Function< List< TypeLnk< builtin_types, 1 > >, If< BinOp< SimpleVar< 5, 0 >,
   ConstInt< 0 >, GreatThan >, ExpList< FuncCall< 4, 0, List< BinOp< SimpleVar< 5,
   0 >, ConstInt< 10 >, Divide > > >, ExpList< FuncCall< builtin_funcs, 0, List<
   FuncCall< builtin_funcs, 4, List< BinOp< BinOp< SimpleVar< 5, 0 >, BinOp< BinOp<
    SimpleVar< 5, 0 >, ConstInt< 10 >, Divide >, ConstInt< 10 >, Times >, Minus >,
   FuncCall< builtin_funcs, 3, List< ConstString< 0 > > >, Plus > > > > > > > >, 4
  >
 >,
ExpList< If< BinOp< SimpleVar< 3, 0 >, ConstInt< 0 >, LessThan >, ExpList<
   FuncCall< builtin_funcs, 0, List< ConstString< 1 > > >, ExpList< FuncCall< 4, 0,
    List< BinOp< ConstInt< 0 >, SimpleVar< 3, 0 >, Minus > > > > >, If< BinOp<
   SimpleVar< 3, 0 >, ConstInt< 0 >, GreatThan >, FuncCall< 4, 0, List< SimpleVar<
   3, 0 > > >, FuncCall< builtin_funcs, 0, List< ConstString< 2 > > > > > > >
```

```
, 2 >
, List<
Function< List< TypeLnk< 0, 0 > >, LetInEnd< List< Variable< ConstInt< 0 >,
   builtin_types , 1 > >,
LetInEnd< List<
Function< List< TypeLnk< builtin_types , 2 > >, If< BinOp< FuncCall< builtin_funcs ,
    3, List< SimpleVar< 1, 0 > > >, FuncCall< builtin_funcs , 3, List< ConstString<
   3 > > >, GreatEq >, BinOp< FuncCall< builtin_funcs , 3, List< SimpleVar< 1, 0 > >
    >, FuncCall< builtin_funcs , 3, List< ConstString< 4 > > >, LessEq >, ConstInt<
   0 > >, 5 >
, List<
Function< List< >, While< If< BinOp< SimpleVar< 1, 0 >, ConstString< 5 >, Equal
   >, ConstInt< 1 >, BinOp< SimpleVar< 1, 0 >, ConstString< 6 >, Equal > >, Assign<
    SimpleVar< 1, 0 >, FuncCall< builtin_funcs , 9, List< > > > >, 5 >
 > >,
ExpList< FuncCall< 5, 1, List< > >, ExpList< Assign< FieldVar< SimpleVar< 3, 0 >,
    0 >, FuncCall< 5, 0, List< SimpleVar< 1, 0 > > > > >, ExpList< While< FuncCall<
   5, 0, List< SimpleVar< 1, 0 > > >, ExpList< Assign< SimpleVar< 4, 0 >, BinOp<
   BinOp< BinOp< SimpleVar< 4, 0 >, ConstInt< 10 >, Times >, FuncCall<
   builtin_funcs , 3, List< SimpleVar< 1, 0 > > >, Plus >, FuncCall< builtin_funcs ,
   3, List< ConstString< 7 > > >, Minus > >, ExpList< Assign< SimpleVar< 1, 0 >,
   FuncCall< builtin_funcs , 9, List< > > > > > >, ExpList< SimpleVar< 4, 0 > > > >
    > > >
, 2 >
 > >,
LetInEnd< List< RecordType< List< TypeLnk< builtin_types , 1 >, List< TypeLnk< 3, 0
    > > > > >,
LetInEnd< List<
Function< List< >, LetInEnd< List< Variable< Record< 0, 0, List< ConstInt< 0 > >
   >, 0, 0 >, List< Variable< FuncCall< 2, 1, List< SimpleVar< 6, 0 > > >,
   builtin_types , 1 > > >,
ExpList< If< FieldVar< SimpleVar< 6, 0 >, 0 >, Record< 3, 0, List< SimpleVar< 6, 1
    >, List< FuncCall< 4, 0, List< > > > > >, Nil > > >
, 4 >
, List<
Function< List< TypeLnk< 3, 0 >, List< TypeLnk< 3, 0 > > >, If< BinOp< SimpleVar<
   5, 0 >, Nil , Equal >, SimpleVar< 5, 1 >, If< BinOp< SimpleVar< 5, 1 >, Nil ,
   Equal >, SimpleVar< 5, 0 >, If< BinOp< FieldVar< SimpleVar< 5, 0 >, 0 >,
   FieldVar< SimpleVar< 5, 1 >, 0 >, LessThan >, Record< 3, 0, List< FieldVar<
   SimpleVar< 5, 0 >, 0 >, List< FuncCall< 4, 1, List< FieldVar< SimpleVar< 5, 0 >,
    1 >, List< SimpleVar< 5, 1 > > > > > > >, Record< 3, 0, List< FieldVar<
   SimpleVar< 5, 1 >, 0 >, List< FuncCall< 4, 1, List< SimpleVar< 5, 0 >, List<
   FieldVar< SimpleVar< 5, 1 >, 1 > > > > > > > > > >, 4 >
, List<
Function< List< TypeLnk< 3, 0 > >, If< BinOp< SimpleVar< 5, 0 >, Nil , Equal >,
   FuncCall< builtin_funcs , 0, List< ConstString< 8 > > >, ExpList< FuncCall< 2, 0,
    List< FieldVar< SimpleVar< 5, 0 >, 0 > > >, ExpList< FuncCall< builtin_funcs ,
   0, List< ConstString< 9 > > >, ExpList< FuncCall< 4, 2, List< FieldVar<
   SimpleVar< 5, 0 >, 1 > > > > > > >, 4 >
 > > >,
LetInEnd< List< Variable< FuncCall< 4, 0, List< > >, builtin_types , 0 >, List<
   Variable< ExpList< Assign< SimpleVar< 1, 0 >, FuncCall< builtin_funcs , 9, List<
    > > >, ExpList< FuncCall< 4, 0, List< > > > >, builtin_types , 0 > > >,
ExpList< FuncCall< builtin_funcs , 0, List< ConstString< 10 > > >, ExpList<
   FuncCall< 4, 2, List< SimpleVar< 5, 0 > > >, ExpList< FuncCall< builtin_funcs ,
   0, List< ConstString< 11 > > >, ExpList< FuncCall< 4, 2, List< SimpleVar< 5, 1 >
    > >, ExpList< FuncCall< builtin_funcs , 0, List< ConstString< 12 > > >, ExpList<
    FuncCall< 4, 2, List< FuncCall< 4, 1, List< SimpleVar< 5, 0 >, List< SimpleVar<
    5, 1 > > > > > > > > > > > > > > > > > >

         program_t;

const char*      metasmousse::const_string[] = {"0", "-", "0", "0", "9", "␣", "\012
   ", "0", "\012", "␣", "list␣1␣:␣\012", "list␣2␣:␣\012", "merged␣list␣:␣\012",
   NULL};

int main()
{
  return (int)program_t::eval< initial_env_t >::doit();
}
```

JSetL: Declarative Programming in Java with Sets

Elisabetta Poleo and Gianfranco Rossi

Dip. di Matematica, Università di Parma,
Via M. D'Azeglio 85/A, 43100 Parma (Italy)
gianfranco.rossi@unipr.it

Abstract. In this paper we present a Java library—called JSetL—that offers a number of facilities to support declarative programming like those usually found in logic or functional declarative languages: logical variables, list and set data structures (possibly partially specified), unification and constraint solving over sets, nondeterminism. The paper describes the main features of JSetL and it shows, through a number of simple examples, how these features can be exploited to support a real declarative programming style in Java.

Keywords: Declarative Programming; Constraint Programming; Java; Nondeterminism.

1 Introduction

Declarative programming (DP) is usually called into play in the context of functional and logic programming languages (e.g., Haskell and Prolog). Intuitively, declarative programming means focusing on *what* a program does, rather than on *how* it does. Notions such as logical variables, side-effect freeness, functional composition, recursion, nondeterminism, etc., are all valuable features of a programming language that supports declarative programming. High-level of control and data abstractions, as well as a clear semantics, are also fundamental features to support the declarative reading of a program. Declarative programming have been mainly exploited in artificial intelligence and automated reasoning applications, but most of its features can be conveniently used also in more general settings to support rapid software prototyping and (automatic) program verification, as well as to allow parallel execution of programs.

Declarative programming is often associated with *constraint programming* (CP), both in the context of logic programming languages (e.g., ECLIPSE [9]), and in the context of functional and functional plus logic programming languages (e.g., Oz [13])). As a matter of fact, constraints provide a powerful tool for stating solutions as sets of equations and disequations over the selected domains, which are then solved by using domain specific knowledge, with no concern to the order in which they occur in the program. As such, CP languages constitute powerful modelling tool, in particularly suitable to coincisely express solutions for artificial intelligence and constraint-satisfaction problems (e.g, combinatorial problems).

While it is undeniable that DP languages (and, in particular, CP languages) provide valuable support for programming, it is also a reality that most real-world software development is still done using traditional, possibly object-oriented (OO), programming languages, such as C++ and Java.

Efforts to make DP languages more appealing for real-world applications have led to various proposals, mainly intended to include object-oriented features into DP languages: several languages, like Prolog and Haskell, have indeed object-oriented extensions. A complementary approach is trying to embed DP features in a more conventional framework—in particular an object-oriented one—in which one can exploit the DP paradigm while retaining all the advantages of constructs for programming and software structuring that are typical of conventional programming languages. This integration can take place according to (at least) two distinct approaches: (i) making the new features available as part of a *library* for some existing language; (ii) defining a new programming language, or extending an existing one, in such a way DP features are viewed as “first-class citizens” of the language itself. Both approaches have pros and cons and a precise comparison of them is likely to be an interesting topic for future research.

One of the best known proposals that integrate some DP features in a conventional OO framework following the *library approach* is that of the ILOG Solver [14, 12]. In this system, constraints and logical variables are handled as objects and are defined within a C++ class library. Thanks to the encapsulation and operator overloading mechanisms, programmers can view constraints almost as if they really were part of the language. Among other proposals that take a similar approach we can mention INC++ [11], NeMo+ [16], and JSolver [2], as concerns the addition of constraints to OO languages, while Frappè [3] and Gisela [10] are two proposals that face the more general problem of making declarative programming features available in a conventional programming environment, though focusing on some specific applications. Another proposal that can be cited in this context is `tu`Prolog [5], a Java package that implements Prolog.

The *new language approach*, in the context of conventional programming languages (approach (ii)), is adopted for instance in the language Alma-0 [1], in Singleton [15], and in DJ (Declarative Java) [17, 18]. A potential advantage of this approach with respect to that based on a library is that it allows a tighter integration between constructs of the host language and DP facilities, making programs simpler and more “natural” to write. On the other hand, however, the design and development of a new language is surely a more difficult task, and the resulting systems are likely to be less easy to integrate with other existing systems and to be accepted by programmers.

The work presented in this paper is another proposal following the OO library approach: we endow an OO language, namely Java, with facilities for supporting declarative programming, by providing them as a library—called JSetL. Differently from other related work we do not restrict ourselves to constraints, but we try to provide a more comprehensive collection of facilities to support a real declarative style of programming. Furthermore, we try to keep our proposal as general as possible, to provide a general-purpose tool not devoted to any specific application. The most notable features of JSetL are:

- logical variables;
- list and set data structures, possibly partially specified (i.e., containing uninitialized logical variables)

- unification (in particular, unification over lists and sets)
- a powerful set constraint solver which allows to compute with partially specified data
- nondeterminism (though confined to constraint solving).

We claim that these facilities provide a valuable support to *declarative programming* and we show this with a number of simple examples. In particular the constraint solver allows complex (set) expressions to be checked for satisfiability, disregarding their order and the instantiation of (logical) variables occurring in them. Moreover, the use of partially specified data structures, along with the nondeterminism “naturally” supported by operations over sets, are fundamental features to allow the language to be used as a highly declarative modelling tool.

The paper is organized as follows. In Section 2 we give an informal presentation of JSetL by showing a simple Java program using JSetL. In Section 3 we introduce the fundamental data structures of JSetL, namely logical variables, sets and lists. In Section 4, we describe the (set) constraint handling facilities supported by our library and we show how constraint solving can be accomplished, and how it interacts with the usual notion of program computation. The fundamental notion of nondeterminism and its relationship with sets is addressed in Section 5. In Section 6 we show how user defined constraints can be introduced in a program and how they can be used. Finally, in Section 7 we briefly discuss future work.

2 An informal introduction to programming with JSetL

First of all we show a simple example of a Java program using JSetL which allows us to give the flavor of the programming style supported by the library.

Problem: Compute and print the maximum of a set of integers `s`.

A truly declarative solution for this problem can be stated as follows: an element `x` of `s` is the maximum of `s`, if for each element `y` of `s` it holds that `y` ≤ `x`. The program below shows how this solution can be immediately implemented in Java using JSetL. Observe that here we are deliberately assuming that execution is not a primary requirement. Indeed, JSetL is mainly conceived as a tool for rapid software prototyping, where easiness of program development and program understanding prevail over efficiency.

```
class Max
    {
    public static Lvar max(Set s) throws Failure
        {
        Lvar x = new Lvar();
        Lvar y = new Lvar();
        Solver.add(x.in(s));
        Solver.forall(y,s,y.leq(x));
        Solver.solve();
        return x;
        }
    public static void main (String[] args)
    throws IOException, Failure
        {
        int[] sample_set_elems = {1,6,4,8,10,5};
        Set sample_set = new Set(sample_set_elems);
        System.out.print(" Max = ");
        max(sample_set).print();
        }
    }
```

For the sake of simplicity we assume that the set of integers is directly supplied by the program (instead of being read for instance from a file). Hence we will focus on the definition of the method `max` that computes the maximum of `s`. `x` and `y` in `max` are two logical variables and both are uninitialized. Invocation of the `add` method adds the constraint `x.in(s)` (i.e., $\mathtt{x} \in \mathtt{s}$) to the current constraint store. This constraint is evaluated to true if `s` is a set and `x` belongs to `s`. If `x` is uninitialized when the expression is evaluated this amounts to *nondeterministically* assign an element of `s` to `x`. Invocation of the `forall` method allows us to add to the constraint store a new constraint `y.leq(x)` (i.e., $\mathtt{y} \leq \mathtt{x}$) for each `y` belonging to `s`. As soon as the `solve` method is invoked the constraint solver checks whether the current collection of constraints in the constraint store is satisfiable or not. If it is, the invocation of the `solve` method terminates with success. The value of `x` represents the integer we are looking for and it is returned as the result of `max`. If, on the contrary, one of the constraints in the constraint store is evaluated to false, backtracking takes place and the computation goes back till the nearest choice point. In this case, the nearest and only choice point is the one created by the `x.in(s)` constraint. Its execution will bind nondeterministically `x` to each element of `s`, one after the other. If all values of `s` have been attempted, there is no further alternative to explore and the computation of `max` terminates raising an exception `Failure`. If no **catch** clause for this exception is provided, the whole computation terminates reporting a failure (actually this is not the case of the `max` method, since a value of `x` for which all the constraints hold surely exists—exactly the maximum of `s`).

Executing the program with the sample set of integers declared in the **main** method causes the message `Max = 10` to be printed to the standard output.

3 Logical variables and composite data objects

JSetL provides logical variables and two new kinds of data structures: sets and lists. These new features are implemented by three classes, `Lvar`, `Lst`, and `Set`, for creation and manipulation of logical variables, lists and sets, respectively.

Lists and sets represent two different data abstractions: while in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter. Thus, for instance, $\{1,2\}$, $\{2,1\}$, and $\{2,1,2\}$ all denote the same set, while the analogous list expressions denote different lists. Although sets and lists can be often used interchangeably, there are cases—especially when sets and lists contain unknown elements—in which one choice can be more appropriate than the other one. For example, if one wants to represent an undirected graph, arcs are likely to be represented as sets rather than as lists. As another example, if one wants to state that a collection C must contain the number 1, disregarding the position of 1 in C, C is conveniently represented as a partially specified set (cf., e.g., set `s2` in Example 2).

3.1 Logical variables

A *logical variable* is an instance of the class `Lvar`, created by the statement

```
Lvar VarName = new Lvar(VarNameExt,VarValue);
```

where VarName is the variable name, VarNameExt is an optional external name of the variable, and VarValue is an optional `Lvar` value associated with the variable.

The *external name* is a string value which can be useful when printing the variable and the possible constraints involving it (if omitted, a default name of the form `"Lvar_`n`"`, where n is a unique integer, is assigned to the variable automatically). `Lvar` are not typed. An *`Lvar` value* can be either a primitive type value, or any library or user defined class object (provided it supplies a method `equals` for testing equality between two instances of the class itself). In particular, an `Lvar` value can be an instance of `Lvar`, `Lst`, or `Set`.

A logical variable which has no `Lvar` value associated with it, or whose `Lvar` value is an uninitialized logical variable (or an uninitialized list or set), is said to be *uninitialized* (or an *unknown*). Otherwise, the logical variable is *initialized*. `Lvar` values other than uninitialized logical variables (or lists or sets) are said *known values*. Uninitialized logical variables will possibly assume a known value (i.e., they become initialized) during the computation, in consequence of some constraints involving them.

3.2 List and set definitions

A *list* is a finite (possibly empty) sequence of arbitrary `Lvar` values (i.e., the *elements* of the list). In JSetL a list is an instance of the class `Lst`, created by the statement

```
Lst LstName = new Lst(LstNameExt,LstElemValues);
```

where LstName is the list name, LstNameExt is an optional *external name* of the list, and VarElemValues is an optional array of `Lvar` values $c_1, \ldots, c_n$ of type t, which constitute the elements of the list. The constant `Lst.empty` is used to denote the *empty list*. No typing information on elements of a list are provided.

A list can be either initialized or uninitialized. An uninitialized list is like a logical variable, but constrained to be (possibly) initialized by list objects only.

A *set* is a finite (possibly empty) collection of arbitrary `Lvar` values (i.e., the *elements* of the set). In JSetL a set is an instance of the class `Set`, created by the statement

```
Set SetName = new Set(SetNameExt,SetElemValues);
```

where SetName, SetNameExt, and SetElemValues have the same meaning than in lists. The constant `Set.empty` is used to denote the *empty set*. Like a list, a set can be either initialized or uninitialized, and no typing information are associated with the elements of a set. Differences between lists and sets become evident when operating on them through list/set operations (e.g., list/set unification—see Sect. 4).

Example 1 `Lvar`, `Lst`, *and* `Set` *definitions*

```
Lvar x = new Lvar();           // uninitialized l. var.
Lvar y = new Lvar("y",'a');    // initialized l. var.
                               // (value 'a'),
                               // with ext'l name "y"
Lvar t = new Lvar(x);          // uninitialized l. var.
                               // (same as variable x)
Lst l = new Lst("l");          // uninitialized list,
                               // with ext'l name "l"
int[] s_elems = {2,4,8,3};
Set s = new Set("s",s_elems); // initialized set
                               // (value {2,4,8,3}),
                               // with ext'l name "s"
```

Hereafter, we will often make use of an abstract notation—which closely resembles that of Prolog—to write lists in a more convenient way. Specifically, $[e_1, e_2, \ldots, e_n]$ is used to denote the list containing n elements e_1, e_2, ..., e_n, while $[\,]$ is used to denote the *empty list*. Moreover, $[e_1, e_2, \ldots, e_n \mid R]$, where R is a list, is used to denote a list containing the n elements e_1, e_2, ..., e_n, plus elements in R. In particular, if R is uninitialized, $[e_1, e_2, \ldots, e_n \mid R]$ represents an "unbounded" list, with elements e_1, ..., e_n and an unknown part R. Similar abstract notation will be introduced also to represent sets (with square brackets replaced by curly brackets).

Elements of a list or of a set can be also logical variables (or lists or sets), possibly uninitialized. For example, the following declarations

```
Lvar x = new Lvar();
Object[] pl_elems = {new Integer(1),x};
Lst pl = new Lst(pl_elems);
```

create the list `pl` with value `[1,x]`, where `x` is an uninitialized logical variable. A list (resp., set) that contains some elements which are uninitialized logical variables (or lists, or sets) is said a *partially specified list (set)*. Note that in a partially specified set the cardinality is not completely determined. For example, the partially specified set {`1,x`} has cardinality 1 or 2 depending on whether `x` will get value `1` or different from `1`, respectively (actually, each partially specified set/list denotes a possibly infinite collection of different sets/lists, that is all sets/lists which can be obtained by assigning admissible values to the uninitialized variables).

3.3 List and set constructor expressions

A list (resp., set) can be also obtained as the result of evaluating a list (resp., set) constructor expression.

Let e be an *`Lvar` expression* (i.e. an expression returning a `Lvar` value), `l` and `m` be *list expressions* (i.e., expressions returning a list object or a logical variable whose value is a list object), and `x` be an uninstantiated logical variable. A *list constructor* is an expression of one of the forms:

(i) l.ins1(e) (head element insertion)
(ii) l.insn(e) (tail element insertion)
(iii) l.ext1(x) (head element removal)
(iv) l.extn(x) (tail element removal)

Expressions (i) and (ii) denote the list obtained by adding $val(e)$ as the first and the last element of the list l, respectively, whereas expressions (iii) and (iv) denote the list obtained by removing from l the first and the last element, respectively. Evaluation of expressions (iii) and (iv) also causes the value of the removed element to become the value of x. [1] It is important to notice that these methods do not modify the list on which they are invoked: rather they build and return a new list obtained by adding/removing the elements to/from the input list (the same will hold for sets, too).

Constructor expressions for sets are simpler than those for lists. In fact, in lists we can distinguish between the first (the *head*) and the last (the *tail*) element of a list, while in sets the order of elements is immaterial. Moreover, only the element insertion method is provided since element extraction may involve a nondeterministic selection of the element to be extracted that is better handled using set constraints (see Section 4). Let e be an Lvar expression and s be a *set expression* (i.e., an expression returning a set object or a logical variable whose value is a set object). A *set constructor* is an expression of the form:

s.ins(e) (element insertion)

which denotes the set obtained by adding $val(e)$ to s (i.e., $\mathtt{s} \cup \{val(e)\}$).

Set/List insertion and extraction methods can be concatenated (left associative). In fact these methods always return a Set/Lst object, and the returned object can be used as the invocation object as well.

Using the insertion methods it is possible to build *unbounded* partially specified sets/lists, that is data structures with a certain number of (either known or unknown) elements $e_1, ..., e_n$, and an unknown "rest" part, represented by an uninitialized set/list r (i.e., using the abstract notation, $\{e_1, \ldots, e_n \mid r\}$ or $[e_1, \ldots, e_n \mid r]$ for sets and lists, respectively).

Example 2 *Set/List element insertion and removal*

```
Lst nil = Lst.empty;            // the empty list
Lst l1 = nil.ins1(3+2).ins1(x);
                                // the p.s. list [x,5]
                                // (x uninitialized var.)
Lst l2 = l1.ext1(y).insn(y);
                                // the p.s. list [5,x]
                                // (y uninitialized var.)
Set s1 = Set.empty.ins(1).ins('a');
                                // the set {'a',1}
Set r = new Set();            // an uninitialized set
Set s2 = r.ins(1);              // the unbounded set {1 | r}
```

[1] Extraction methods for lists require that the invocation list l is initialized and that x is not initialized. If one of these conditions is not respected an exception is raised (namely, NotInitVarException and InitLvarException, respectively). Moreover, if l is the empty list, a EmptyLstException exception is raised.

Note that `s2` in the above example is a partially specified set containing one element, `1`, and an unknown part `r`; in this case, the cardinality of the denoted set has no upper bound (the lower being 1).

Special forms of the insertion and extraction methods are provided to simplify their usage. In particular, the method `ins1All(a)`, applied to a list `l`, where `a` is an array of elements of a type *t*, returns a list obtained from `l` by adding all elements of `a` as the head elements of `l`, respecting the order they have in `a`. Similarly, `insAll(a)`, applied to a set `s`, is used to insert more than one element at a time into `s`. In addition, an alternative form is provided for specifying the value for a set or list object. When creating the object it is possible to specify the limits l and u of an interval $[l, u]$ of integers: the elements of the interval will be the elements of the set/list (if $u < l$ the set/list is empty).

A number of utility methods are also provided by classes `Lvar`, `Lst`, and `Set`. These methods are used, for example, to print a set/list object, to know whether a logical variable is initialized or not, to get the external name associated with a `Lvar`, `Lst`, or `Set` object, and so on.

Logical variables, sets, and lists are used mainly in conjunction with constraints. Constraints are addressed in more details in the next section.

4 Programming with (Set) Constraints

Basic set-theoretical operations, as well as equalities and inequalities, are dealt with as *constraints* in JSetL. The evaluation of expressions containing such operations is carried on in the context of the current collection of active constraints $\mathcal{C}$ (the global *constraint store*) using domain specific constraint solvers. Those parts of these expressions, usually involving one or more uninitialized variables, which cannot be completely solved are added to the constraint store and will be used to narrow the set of possible values that can be assigned to the uninitialized variables.

4.1 JSetL constraints

The JSetL *constraint domain* is the $\mathcal{SET}$ domain defined in [6], extended with a few new constraints over lists and integers. An *atomic constraint* of this domain is an expression of one of the forms:

- $e_1.op\,(e_2)$
- $e_1.op\,(e_2, e_3)$

where op is one of a collection of predefined methods provided by classes `Lvar`, `Lst` and `Set`, and e_1, e_2 and e_3 are expressions whose type depends on op. More precisely, let l, r be `Lvar` expressions, s, s_1, s_2, s_3 set expressions, l_1, l_2 list expressions, and i_1, i_2 integer expressions. JSetL provides the following atomic constraints:
$l.\mathsf{eq}\,(r)$ (equality) for comparing `Lvar` values;
$l.\mathsf{in}\,(s)$ (membership), $s_1.\mathsf{subset}\,(s_2)$ (subset), $s_1.\mathsf{union}\,(s_2, s_3)$ (union, i.e. $val(s_1) = val(s_2) \cup val(s_3)$)), $s_1.\mathsf{inters}\,(s_2, s_3)$ (intersection), and a few other basic set-theoretic

operations, for dealing with sets;
$i_1.\mathsf{le}\,(i_2)$ ($\leq$), $i_1.\mathsf{ge}\,(i_2)$ ($\geq$), ..., for comparing integer values.
Moreover, for most of them, also their negative counterparts are provided: $l.\mathsf{neq}\,(r)$ (inequality), $l.\mathsf{nin}\,(s)$ (not membership), $s_1.\mathsf{nsubset}\,(s_2)$ (not subset), and so on.

If an expression e_i of an atomic constraint $e_1.op\,(e_2, e_3)$ (or $e_1.op\,(e_2)$) is evaluated to a value of the wrong type, a suitable exception is raised by the constraint solver.

A *constraint* is either an atomic constraint or (recursively) the conjunction of two or more atomic constraints c_1, c_2,..., c_n:

- $c_1.\mathsf{and}\,(c_2)\ \ldots\ .\mathsf{and}\,(c_n)$

Example 3 *JSetL constraints*

Let x, y, z *be logical variables and* r, s, *and* t *be sets.*

```
r.eq(s);                    // equality between sets
t.union(r,s);               // t = r ∪ s
x.eq(y).and(x.eq(3)).and(y.neq(z))
                            // x = y ∧ x = 3 ∧ y ≠ z
```

Note that solving an equality constraint implies the ability to solve a *set unification* problem (cf., e.g., [7]). Set unification of two (possibly partially specified) sets s and r means finding an assignment of values to uninitialized variables occurring in them (if any), such that s and r become equal in the underlying set theory. Intuitively, in any reasonable set theory, two sets are equal if they have the same elements, disregarding their order and possible repetitions. Thus, for instance, the set unification problem

$$\{x, y\} = \{1, 2\}$$

where x and y are uninitialized logical variables, admits two solutions: assign 1 to x and 2 to y, or assign 2 to x and 1 to y.

4.2 Constraint solving

The approach adopted for constraint solving in JSetL is the one developed for CLP($\mathcal{SET}$) [6]. Logically, the constraint store is a conjunction of atomic formulae built using basic set-theoretic operators, along with equality and inequalitie. Satisfiability is checked in a set-theoretic domain, using a suitable constraint solver which tries to reduce any conjunction of atomic constraints to a simplified form—the *solved form*—which is guaranteed to be satisfiable. The success of this reduction process allows one to conclude the satisfiability of the original collection of constraints. Conversely, the detection of a failure (logically, the reduction to false) implies the unsatisfiability of the original constraints. Solved form constraints are left in the current constraint store and passed ahead to the new state. A successful computation, therefore, may terminate with a not empty collection of solved form constraints in the final constraint store.

The JSetL constraint solver basically implements in Java the constraint solver of CLP($\mathcal{SET}$), extended with simple constraints over integers. Suitable restrictions, however, are imposed on the latter so that they can be always eliminated. Namely, expressions e_1 and e_2 in $e_1.op\,(e_2)$, where op is an integer comparison operator, can not contain any uninstantiated variable when they are evaluated; otherwise an exception is raised.[2] Therefore, since the constraint solver of CLP($\mathcal{SET}$) is proved to be complete (as well as correct and terminating) [6], the same holds also for the JSetL constraint solver.

To add a constraint C to the constraint store, the `add` method of the `Solver` class can be called as follows:

```
Solver.add(C)
```

The order in which constraints are added to the constraint store is completely immaterial. After constraints have been added to the store, one can invoke their resolution by calling the `solve` method:

```
Solver.solve()
```

The `solve` method nondeterministically searches for a solution that satisfies all constraints introduced in the constraint store. If there is no solution a `Failure` exception is generated. We say that the invocation of a method, calling (directly or indirectly) the `solve` method, terminates with *failure* if its execution causes the `Failure` exception to be raised; otherwise we say that it terminates with *success*. The default action for this exception is the immediate termination of the current thread. The exception, however, can be caught by the program and dealt with as preferred.

To find a solution, the constraint solver tries to reduce the atomic constraints in the constraint store to a simplified form - called the *solved form* (see [6]). This reduction is *nondeterministic*. Nondeterminism is handled through choice points and backtracking. Once the constraint reduction process detects a failure, the computation backtracks to the most recently created choice point (chronological backtracking). If no choice point is left open the whole reduction process fails (i.e., the `Failure` exception is generated).

Example 4 *Constraint solving*

Let `s` *be the set* $\{$`x,y,z`$\}$, *where* `x`, `y`, `z` *are uninitialized logical variables, and* `r` *be the set* $\{$`1,2,3`$\}$.

```
Solver.add(r.eq(s));    // set unification r = s
Solver.add(x.neq(1));   // x ≠ 1
Solver.solve();         // calling the constraint solver
x.output();
```

`x.output()` *prints the (external) name of the variable* `x` *followed by its value (if any; otherwise, followed by* `'_'`*). Therefore the output generated by this code fragment is:*

```
x = 2
```

[2] Actually, the new version of the CLP($\mathcal{SET}$) solver [4], which integrates the $\mathcal{SET}$ solver with an efficient solver over *finite domains* ($\mathcal{FD}$), is able to deal with basic operations over integers as constraints with almost no restriction on the instantiation of expressions that can occur in them. Following the same approach, and extending the JSetL constraint solver accordingly, it would allow us to deal with arithmetic constraints with no restrictions also in JSetL.

In the above example, the value for `x` is computed through backtracking. Solving the set unification problem $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{\mathtt{1}, \mathtt{2}, \mathtt{3}\}$ nondeterministically returns one of the six different solutions:

`x = 1, y = 2, z = 3`,
`x = 1, y = 3, z = 2`,
`x = 2, y = 3, z = 1`, ...

and so on. Assuming the first computed value for `x` is `1`, then the other constraint, `x.neq(1)`, turns out to be not satisfied. Thus, backtracking forces the solver to find another solution for `x`, namely `x = 2`. In this case, the conjunction of the two given constraints is satisfied, and the invocation of the `solve` method terminates successfully. If later on a new constraint, e.g., `[x]` $\neq$ `[2]`, is added to the constraint store, and the constraint solver is called again:

```
Solver.add(Lst.empty.ins1(x).neq(Lst.empty.ins1(2)));
                                               // [x] ≠ [2]
Solver.solve();
x.output();
```

the choice points left open by the previous call to the solver are still open and they are explored by the new invocation. The output generated at the end of the computation of this new fragment of code is therefore:

```
x = 3
```

Note that every time the `solve` method is invoked it does not restart solving the constraint from the beginning but it restart from the point reached by the last invocation to `solve`.

At the end of the computation the constraint store may contain solved form constraints. To print these constraints, other than equality constraints, one can use the static method `showStore()` of class `Solver` (actually this method allows to visualize the content of the constraint store at any moment during the computation).

JSetL provides also a more convenient way to introduce more than one constraint at a time, through the `forall` method. Let `x` be an uninitialized variable, S a set expression which is evaluated to a bounded set, C a constraint containing `x`, and C_s the constraint obtained from C by replacing all occurrences of `x` with element s of S. The statement

`Solver.forall(x,`S`,`C`)`

adds the constraint C_s to the constraint store, for each element s of S. Logically, `forall(x,`S`,`C`)` is the so-called Restricted Universal Quantifier (cf., e.g., [6]): $\forall x((x \in S) \rightarrow C)$ (see the sample program in Section 2 for a simple use of `forall`).

It is common also to allow *local* variables $y_1, \ldots, y_n$ in C, which are created as new for each element of the set (logically, $\forall x((x \in S) \rightarrow \exists y_1, \ldots, y_n(C))$ that is $y_1, \ldots, y_n$ are existentially quantified variables). For this purpose, JSetL provides also the method

`Solver.forall(x,`S`,`Y`,`C`)`

where `x`, S, and C are the same as in the simpler `forall` method, while Y is an array of all the local uninitialized logical variables $y_1, \ldots, y_n$ occurring in C (see Example 6).

4.3 Programming with constraints

Let us see how the solver works on a number of simple examples, possibly involving also constraints in the computed result.

Example 5 *In difference*

Check whether an element `x` *belongs to the difference between two sets,* `s1` *and* `s2` *(i.e.,* $\mathtt{x} \in \mathtt{s1} \setminus \mathtt{s2}$*) .*

```
public static void in_difference(Lvar x, Set s1, Set s2)
throws Failure
   {
   Solver.add(x.in(s1));
   Solver.add(x.nin(s2));
   Solver.solve();
   }
```

If the following code fragment is executed (for instance, in the **main** *method)*

```
in_difference(x,s,r);
x.output();
Solver.showStore();
```

and `s` *and* `r` *are the sets* {`1,2`} *and* {`1,3`}*, respectively, and* `x` *is an uninitialized variable, the output generated is:*

```
x = 2
Store: empty
```

Conversely, if `s` *is an uninitialized set, then executing the same program fragment as above, will produce the following output*

```
x = unknown
s = {x | Set_1}
Store: x.neq(1) x.neq(3)
```

which is read as: `s` *can be any set containing the element* `x` *and* `x` *must be different from* `1` *and* `3`*.*

The ability to solve constraints disregarding the fact logical variables occurring in them are initialized or not allows methods involving constraints to be used in a quite flexible way, e.g., using the same method both for testing and computing solutions This flexibility strongly contributes to support a declarative programming style.

Example 6 *All pairs*

Check whether all elements of a set `s` *are pairs, i.e., they have the form* `[x1,x2]`*, for any* `x1` *and* `x2`*.*

```
public static void all_pairs(Set s) throws Failure
   {
   Lvar x1 = new Lvar();
   Lvar x2 = new Lvar();
   Lvar[] Y = {x1,x2};
   Lvar x = new Lvar();
   Lst pair = Lst.empty.ins1(x2).ins1(x1);
   Solver.forall(x,s,Y,x.eq(pair));
   Solver.solve();
   return;
   }
```

Let `sample_set` *be the set* {`[1,3],[1,2],[2,3]`}. *The following fragment of code tests whether* `sample_set` *is composed only of pairs and prints a message* `‘‘All pairs’’` *or* `‘‘Not all pairs’’` *depending on the result of the test.*

```
boolean res = true;
try {
   all_pairs(sample_set);
   }
catch(Failure e)
   {res = false;}
if (res) System.out.print("All pairs");
else System.out.print("Not all pairs");
```

Example 6 shows also how a statement, namely `all_pairs(sample_set)`, can be used, in a sense, as a condition. In fact, if execution of the statement fails (i.e., not all elements in the given set are pairs), then an exception `Failure` is raised and the associated exception handler executed. The latter can easily set a boolean variable to be used in the next **if** statement. Thus, if the statement terminates with success then a `true` value is returned (in `res`); otherwise, the statement terminates with failure and a `false` value is returned. This is analogous to the use of statements as expressions found in some languages, such as Alma-0 [1] and SINGLETON [15].

Constraints and other JSetL facilities can be used in conjunction with the usual control structures of Java. This situation is illustrated by the following example.

Example 7 *Symmetrical list*

Check whether a list `l` *is symmetrical or not.*

```
public static boolean symmetrical(Lst l)
throws Failure
   {
   try {
      while(l.size()>1)
         {
         Lvar z1 = new Lvar();
         Lvar z2 = new Lvar();
         Lst r = l.ext1(z1).extn(z2);
            // extract the first and last element of l
         Solver.add(z1.eq(z2));
            // the first and the last elem’s
            // must be equal
         Solver.solve();
         l = r;
            // continue with the rest of l
         }
      return true;
      }
   catch(Failure e)
      {
      return false;
      }
   }
```

If, for example, `l` *is* `['r','a','d','a','r']` *the value returned by* `symmetrical` *is* **true**. *List* `l` *can contain also some unknown values. For example, with* `l` = `[x,1,3,y,2]`, `x` *and* `y` *uninitialized variables, invocation of the* `symmetrical` *method returns* **true** *and as a side-effect it initializes variables* `x` *and* `y` *to* `2` *and* `1`, *respectively. Note that within the* **while** *loop we use an assignment between two logical variables,* `l = r`*: this forces* `l` *at the next iteration to be replaced by the new (shorter) list* `r`.

5 Nondeterminism

A computation in JSetL can be nondeterministic, though nondeterminism in JSetL is confined to constraint solving. Precisely, like in SINGLETON, nondeterminism is mainly supported by set operations. As a matter of fact, the notion of nondeterminism fits into that of set very naturally. Set unification and many other set operations are inherently and naturally nondeterministic. For example, the evaluation of $x \in \{1,2,3\}$ with x an uninitialized variable, nondeterministically returns one among $x = 1$, $x = 2$, $x = 3$. Since the semantics of set operations is usually well understood and quite "intuitive", making nondeterministic programming the same as programming with sets can contribute to make the (not trivial) notion of nondeterminism easier to understand and to use.

Nondeterminism is another key feature of a programming language to support declarative programming. A simple way to exploit nondeterminism in JSetL is through the use of the `Setof` method. This method allows one to explore the whole search space of a nondeterministic computation and to collect into a set all the computed solutions for a specified logical variable `x`. Then the collected set can be processed, e.g., by iterating over all its elements using the `forall` method.

Example 8 *All solutions*

Compute the set of all subsets (i.e., the powerset) of a given set `s`.

```
   public static Set powerset(Set s) throws Failure
      {
      Set r = new Set();
      Solver.add(r.subset(s));
      return Solver.setof(r);
      }
```

If `s` *is the set* {`'a','b'`}, *the set returned by* `powerset` *is* {{},{`'a'`},{`'b'`}, {`'a','b'`}}.

As a more comprehensive example, using nondeterminism and set constraints, we show a possible JSetL solution to the well-known combinatorial problem of the coloring of a map.

Example 9 *Coloring of a map*

Given a map of n regions r_1,...,r_n and a set of m colors c_1,...c_m find an assignment of colors to regions such that neighboring regions have different colors.

The regions are represented by a set of n uninitialized logical variables and the colors by a set of m constant values (e.g., {`"red","blue"`}*). The map is modeled by an undirected graph and it is represented as a set whose elements are sets containing two neighboring regions. At the end of the computation each* `Lvar` *representing a region will be initialized with one of the given color.*

```
public static void coloring(Set regions,
                            Set map,
                            Set colors)
throws Failure
   {
   Lvar x = new Lvar();
   Set single = Set.empty.ins(x);
   Solver.add(regions.eq(colors));
   Solver.forall(x,colors,(single.nin(map));
   Solver.solve();
   return;
   }
```

The solution uses a pure "generate & test" approach. The `regions` = `colors` *constraint allows us to find a valuable assignment of colors to regions. Invocation of the* `forall` *method allows us to test whether the constraint* {`x`} $\notin$ `map` *holds for all* `x` *belonging to* `colors`*. If it holds, it means that for no pair* $\{r_i, r_j\}$ *in* `map`*, r_i and r_j have got the same color.*

If `coloring` *is called with* `regions` = {`r1,r2,r3`}*,* `r1`*,* `r2`*,* `r3` *uninitialized logical variables,* `map` = {{`r1,r2`},{`r2,r3`}}*, and* `colors` = {`"red", "blue"`}*, the invocation terminates with success, and* `r1`*,* `r2`*,* `r3` *are initialized to* `"red"`*,* `"blue"`*, and* `"red"`*, respectively (actually, also the other solution which initializes* `r1`*,* `r2`*,* `r3` *to* `"blue"`*,* `"red"`*, and* `"blue"`*, respectively, can be computed through backtracking, if the first computed solution turns out to cause a failure).*

Note that the set of colors can be also partially specified. For example, if `colors` = {`c1,"blue"`}*, with* `c1` *an uninitialized variable, executing* `coloring` *will generate the constraint:*
`r1` = `Lvar_1`, `r2` = `blue`, `r3` = `Lvar_1`, `Lvar_1.neq(blue)`.

6 Defining new constraints

Nondeterminism in JSetL is confined to constraint solving. One consequence of this is that backtracking allows the computation to go back to the nearest open choice point within the constraint solver, but it does not allow to “re-execute” user program code. For example, let us consider the following program fragment, where we assume that `s` is the set $\{$`0,1`$\}$, and c_1, c_2 are two constraints:

```
Solver.add(x.in(s));
Solver.solve();
if (x.value().equals(new Integer(0)) Solver.add(c1);
else Solver.add(c2);
```

If, when evaluating the **if** condition, the value of the logical variable `x` is `0` then the constraint c_1 is added to the constraint store. If, subsequently, a failure is detected, backtracking will allow to consider a different value for `x`, namely `1`, but the **if** condition is no longer evaluated. The constraint solver will examine the constraint store again, with the new value for `x` but still with constraint c_1 added to it.

Basically the problem is caused by the fact that we cannot guarantee a tight integration between the constraint solver (which is defined in a library) and the primitive constructs of the language. This is probably the main difference between what we called the “library” approach and the approach based on the definition of a new language (or the extension of an existing one). As a matter of fact the problem illustrated by the above program fragment is easily programmed in a language such as SINGLETON where nondeterminism and logic variables are embedded in the language.

However, JSetL provides a solution to overcome this difficulty. The solution is based on the possibility to introduce user-defined new constraints. Whenever a method which the user wants to define requires some nondeterministic action embedded in a nontrivial control structure, one can define the method as a new constraint, so that its execution is completely performed in the context of constraint solving.

User-defined constraints are defined as usual Java methods except that: (i) they are all declared within a class named `NewConstraints`; (ii) they can use the JSetL facilities for handling nondeterminism. To make the use of these facilities simpler, we assume a special construct is provided (similar to that found for instance in Alma-0 [1]): ***either*** $\texttt{S}_1$ ***orelse*** $\texttt{S}_2$ ... ***orelse*** $\texttt{S}_\texttt{n}$, where $\texttt{S}_1$,...,$\texttt{S}_\texttt{n}$ are Java statements. The logical meaning of the ***either-orelse*** construct is the disjunction $\texttt{S}_1\vee$,...,$\vee$ $\texttt{S}_\texttt{n}$, while its computational interpretation is that of exploring, through backtracking, all possible alternatives $\texttt{S}_1$,...,$\texttt{S}_\texttt{n}$, starting from $\texttt{S}_1$. Actually this construct is easily replaced by usual Java code endowed with JSetL facilities for handling nondeterminism (see Appendix A for an example). This replacement can be performed automatically through a straightforward *preprocessing* phase that takes the extended Java code of the `NewConstraints` class as its input and it generates the corresponding standard Java code.

Let us see how the user can define a new constraint using a simple example: a fully nondeterministic recursive definition of the classical list concatenation operation (`concat`). The solution, however, can be easily generalized to other cases.

```
class NewConstraints
```

```
{
public static void concat(Lst l1, Lst l2, Lst l3)
throws Failure
   {
    either
      {
      Solver.add(l1.eq(Lst.empty));
      Solver.add(l2.eq(l3));
      }
    orelse
      {
      Lvar x = new Lvar();
      Lst l1new = new Lst();
      Lst l3new = new Lst();
      Solver.add(l1.eq(l1new.ins1(x)));
                         // l1 = [x | l1new]
      Solver.add(l3.eq(l3new.ins1(x)));
                         // l3 = [x | l3new]
      Solver.add(concat(l1new,l2,l3new));
                         // concat(l1new,l2,l3new)
      }
   return;
   }
}
```

The method `concat` takes three lists as its parameters: `l1`, `l2` and `l3`. `l3` is the concatenation of `l1` and `l2`. `concat` can be used both to check if a given concatenation of lists holds and to build any of the three lists, given any of the other two (as usual, for instance, in Prolog). Such flexibility is obtained by using unification (instead of standard assignment) and nondeterminism (instead of a deterministic **if** statement). Nondeterminism is implemented through the ***either-orelse*** construct: the first alternative states that when `l1` is the empty list, `l2` and `l3` must be equal; the second alternative deals with the case in which the first element of `l1` is `x` so that `l3` is obtained by inserting `x` as the head element of the list `l3new` which is obtained (recursively) by concatenating the rest of `l1` (i.e., `l1new`) with `l2`. The actual Java code that implements the new constraint `concat` (generated through preprocessing of the `NewConstraints` class) is shown in Appendix A.

Execution of the statement

```
Solver.add(NewConstraints.concat(l1,l2,l3))
```

causes the user-defined constraint `concat` to be added to the current constraint store. If, for instance, `l1` is `[1,2,3]`, `l2` is `[4,5]`, and `l3` is an uninitialized list, a subsequent call to `Solver.solve()` will set `l3` equal to `[1,2,3,4,5]`.

As an example of the possible different usages of `concat` we show a completely declarative version of a method for checking the sublist relation.

Example 10 *Sublist relation*

Check whether a list `p` *is a sublist of a list* `s` *and return the position* `k` *where* `p` *starts.*

```
public static void prefix(Lst l1, Lst l2)
throws Failure
   {
   Lst l = new Lst();
   Solver.add(NewConstraints.concat(l1,l,l2));
   Solver.solve();
   return;
   }

public static int sublist(Lst p, Lst s)
throws Failure
   {
   Lst A = new Lst();
   Lst B = new Lst();
   prefix(A,s);
   Solver.add(NewConstraints.concat(A,p,B));
   prefix(B,s);
   Solver.solve();
   return A.size()+1;
   }
```

where A.size() *yields the number of elements of the list denoted by* A*. The first invocation of the* prefix *method in* sublist *allows us to compute (nondeterministically) a possible prefix* A *of* s*, while the second invocation of the* prefix *method is used to check whether the computed sublist* B *is a prefix of* s*. If this is not the case, a different value for* A *is attempted next.*

As an example, if s *is the list* [1,1,3,1,4,5,1,2,1,1,1,4] *and* p *is the list* [1,2,1] *the invocation of* sublist *will return* 7 *as its result.*

7 Conclusions and future work

We have presented the main features of the JSetL library and we have shown how they can be used to write programs that exhibit a quite good declarative reading, while maintaining all the features of conventional Java programs. In particular we have described the (set) constraint handling facilities supported by our library and we have shown how constraint solving can be accomplished, and how it interacts with the usual notion of program computation. Furthermore we have shown how to exploit nondeterminism, possibly by introducing new used-defined constraints. JSetL is fully implemented in Java and is available at URL www.math.unipr.it/~gianfr/JSetL.

All features provided by JSetL are present also in the CLP($\mathcal{SET}$) language [6], but embedded in a CLP framework. An attempt to "export" these features outside CLP is represented by the definition of the Singleton language [15], a declarative language that combines most of the features considered in this paper with "traditional" features of imperative programming languages, such as the iterative control structures and the block structure of programs. Singleton, however, is a completely new language, with its own syntax and its own semantics. A possible side-effect of the current work on JSetL is to allow us to compare the approach followed in Singleton with the library based approach followed in JSetL, in order to evaluate the gain in the expressive power related to the effort needed to develop the new facilities and the easiness to use them.

As a future work the constraint solving capabilities of JSetL could be strongly enhanced by enlarging the constraint domain from that of sets to that of *finite domains*. Following [4], this enhancement could be obtained by integrating an existing constraint solver for finite domains, possibly written in Java, with the JSetL constraint solver over sets. As shown in [4] this would allow us to have, in many cases, the efficiency of the finite domain solvers, while maintaining the expressive power and flexibility of the set constraint solvers (which in turn is inherited from CLP($\mathcal{SET}$)).

On a different side, another concrete improvement could be obtained by using flexible preprocessing tools for the Java language that would allow us to develop suitable syntax extensions that would make it simpler and more natural using the JSetL facilities.

8 Acknowledgments

The work is partially supported by MIUR projects *Automatic Aggregate—and number—Reasoning for Computing* and *CoVer—Constraint-based Verification of Reactive Systems*.

Bibliography

[1] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM TOPLAS*, 20(5), 1014–1066, 1998.

[2] A.Chun. Constraint programming in Java with JSolver. In *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.

[3] A.Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages, PADL 2001*, LNCS, Vol. 1990, Springer-Verlag, 29–44, 2001.

[4] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. In *PPDP'03 — Proc. of the Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press, 219–229, 2003.

[5] E.Denti, A.Omicini, and A.Ricci. tuProlog: a light-weight Prolog for internet applications and infrastructures. In *Practical Aspects of Declarative Languages, PADL 2001*, LNCS, Vol. 1990, Springer-Verlag, 184–198, 2001.

[6] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.

[7] A. Dovier, E. Pontelli, and G. Rossi. Set unification. TR-CS-001/2001, Dept. of Computer Science, New Mexico State University, USA, January 2001 (available at `www.cs.nmsu.edu/TechReports`).

[8] M.Dincbas, P.Van Hentenryck, H.Simonis, et al. The constraint logic programming CHIP. In *Proc. of the 2nd Int'l Conf. On Fifth Generation Computer Systems*, 683-702, 1988.

[9] ECLiPSe, User Manual. Tech. Rept., Imperial College, London. August 1999. Available at `www.icparc.ic.ac.uk/eclipse`.

[10] G.Falkman, O.Torgersson. Enhancing Usefulness of Declarative Programming Frameworks through Complete Integration. In *Proc. of the 12th Int. Workshop on Logic Programming Environments*, July 2002 (available at `http://xxx.lanl.gov/abs/cs.SE/0207054`).

[11] E.Hyyonen, S.DePascale, and A.Lehtola. Interval constraint satisfaction tool INC++. In *Proc. of the 5th ICTAI*, IEEE Press, 1993.

[12] ILOG Optimisation Suite - White Paper. Available at `www.ilog.com/products/optimisation/tech/optimisation/whitepaper.pdf`.

[13] G.Smolka. The Oz programming model. In *Current Trends in Computer Science*, J. van Leeuwen, Ed., LNCS, vol. 1000, Springer-Verlag, 1995.

[14] J.-F.Puget and M.Leconte. Beyond the Glass Box: Constraints as Objects. In *Proc. of the 1995 Int'l Symposium on Logic Programming*, MIT press, pp. 513-527.

[15] G.Rossi. Set-based Nondeterministic Declarative Programming in SINGLETON. In *11th Int.l Workshop on Functional and (constraint) Logic Programming*, Electronic Notes in Theoretical Computer Science, Vol. 76, Elsevier Science B. V., 17 pages, 2002.

[16] I.Shvetsov, V.Telerman, and D.Ushakov. NeMo+: Object-oriented constraint programming environment based on subdefinite models. In *Artificial Intelligence and Symbolic Mathematical Computations* (G.Smolka, ed.), LNCS 1330, Springer-Verlag, 534-548.

[17] Neng-Fa Zhou. DJ: Declarative Java, Version 0.5, User's manual. Kyushu Institute of Tecnology, 1999. Available at `www.cad.mse.kyutech.ac.jp/people/zhou/dj.htlm`.

[18] Neng-Fa Zhou. Building Java Applets by using DJ—a Java Based Constraint Language. Available at `www.sci.brooklyn.cuny.edu/~zhou`.

A Implementing user-defined constraints

In this section we present the actual code used to implement the user-defined constraint `concat`.

```
class NewConstraints
   {
   public static StoreElem concat(Lst l1, Lst l2, Lst l3)
      {
      StoreElem s = new StoreElem(n,l1,l2,l3);
      return s;
      }

   protected static void user_code(int c, StoreElem s)
   throws Failure
      {
      switch(c)
         { ...
         case n: concat(s); break;
         ...}
      return s;
      }

   public static void concat(StoreElem s)
   throws Failure
      {
      Lst l1 = (Lst)s.arg1;
      Lst l2 = (Lst)s.arg2;
      Lst l3 =(Lst)s.arg3;
      switch(s.caseControl)
         {
         case 0:
            add_ChoicePoint(s);
            add(l1.eq(Lst.empty));
            add(l2.eq(l3));
            return;
         case 1:
            Lvar x = new Lvar();
            Lst l1new = new Lst();
            Lst l3new = new Lst();
            add(l1.eq(l1new.ins1(x)));
                        // l1 = [x | l1new]
            add(l3.eq(l3new.ins1(x)));
                        // l3 = [x | l3new]
            add(concat(l1new,l2,l3new));
                        // concat(l1new,l2,l3new)
            return;
         }
      }
   }
```

The first definition of the concat method creates a new instance of the class StoreElem which is used to store the new constraint within the constraint store. The instance contains all parameters for the concat method, along with an integer n which will be used by the solver to uniquely identify the new constraint. The user_code method is used to associate each internal code with the corresponding method that implements the user-defined constraint (namely, the second definition of the concat method in this example). The control expression of the **switch** statement is the caseControl attribute of the constraint store s associated with concat (default value: 0). Each case block, but the last one, creates a choice point and adds it to the stack of the alternatives by executing the statement add_ChoicePoint(s); then the remaining code of the case block adds the constraints necessary to compute one of the possible solutions.

SML2Java: A Source to Source Translator

Justin Koser, Haakon Larsen, and Jeffrey A. Vaughan

Cornell University

Abstract. Java code is unsafe in several respects. Explicit null references and object downcasting can cause unexpected runtime errors. Java also lacks powerful language features such as pattern matching and first-class functions. However, due to its widespread use, cross-platform compatibility, and comprehensive library support, Java remains a popular language.

This paper discusses SML2Java, a source to source translator. SML2Java operates on type checked SML code and, to the greatest extent possible, produces functionally equivalent Java source. SML2Java allows programmers to combine existing SML code and Java applications.

While direct translation of SML primitive types to Java primitive types is not possible, the Java class system provides a powerful framework for emulating SML value semantics. Function translations are based on a substantial similarity between Java's first-class objects and SML's first-class functions.

1 Introduction

SML2Java is a source-to-source translator from Standard ML (SML), a statically typed functional language [8], to Java, an object-oriented imperative language. A successful translator must emulate distinctive features of one language in the other. For instance, SML's first-class functions are mapped to Java's first-class objects, and an SML let expression could conceivably be translated to a Java interface containing an 'in' function, where every let expression in SML would produce an anonymous instantiation of the let interface in Java. Similarly, many other functional features of SML are translated to take advantage of Java's object-oriented style. Because functional features such as higher-order functions must ultimately be implemented using first-class constructs, we believe one can only achieve a clean design by taking advantage of the strengths of the target language.

SML2Java was inspired by problems encountered teaching functional programming to students familiar with the imperative, object-oriented paradigm. It was developed for possible use as a teaching tool for Cornell's CS 312, a course in functional programming and data structures. For the translator to be a successful educational tool, the translated code must be intuitive for a student with Java experience.

On a broader level, we wish to show how functional concepts can be mapped to object-oriented imperative concepts through a thorough understanding of each model. In this regard, it becomes important not to force functional concepts upon an imperative language, but rather to translate these functional concepts to their imperative equivalents.

2 Translation

This section discusses the choices we made in our translation of SML to Java. Where pertinent, we will also discuss the benefits and drawbacks of our design decisions.

2.1 Primitives

SML primitive types, such as *int* and *string*, are translated to Java classes. The foregoing become *Integer2* and *String2*, respectively. Ideally, SML primitives would translate to their built-in Java equivalents (e.g. *int* → *java.lang.Integer*), but these classes (e.g. *java.lang.Integer*) do not support operations such as integer addition or string concatenation [10]. We do not map directly to *int* and *string* because Java primitives are not derivatives of *Object*, cannot be used with standard Java collections, and are not compatible with our function and record translations. The latter will be shown in sections 2.2 and 2.4. Our classes, which are based on *Java.util.**, include necessary basic operators and fit well with function and record translation. While Hicks [6] addresses differences between the SML and Java type systems, he does not discuss interoperability. Blume [4] treats the related problem of translating C types to SML.

Figure 1 demonstrates a simple translation. The astute reader will notice several superfluous typecasts. Some translated expressions formally return *Object*. However, because SML code is typesafe, we can safely downcast the results of these expressions. The current version of SML2Java is overly conservative, and inserts some unnecessary casts. Additionaly, the add function looks quite complicated. This is consistent with other function translations which are discussed in section 2.4.

2.2 Tuples and Records

We follow the SML/NJ compiler (section 3) which compiles tuples down to records. Thus, every tuple of the form *(exp1, exp2, ...)* becomes a record of the form {*1=exp1, 2=exp2, ...*}. This should not surprise the avid SML fan as SML/NJ will, at the top level, represent the record {*1="hi", 2="bye"*} and the tuple *("hi","bye"): string*string* identically.

The *Record* class represents SML records. Every SML record value maps to an instance of this class in the Java code. The *Record* class contains a private data member, *myMapping*, of type *java.util.HashMap*. SML records are translated to a mapping from fields (which are of type *String*) to the data that they carry (of type *Object*). The *Record* class also contains a function *add*, which takes a *String* and an *Object* as its parameters and adds these to the mapping. A record of length *n* will therefore require *n* calls to *add*. Record projection is little more than a lookup in the record's *HashMap*.

2.3 Datatypes

An SML datatype declaration creates a new type with one or more constructors. Each constructor may be treated as a function of zero or one arguments. SML2Java treats this

Fig. 1. Simple variable binding

SML Code:

```
val x=40
val y=2
val z=x+y
```

Java Equivalent:

```
public class TopLevel {
  public static final Integer2 x = (Integer2)
    (new Integer2 (40));

  public static final Integer2 y = (Integer2)
    (new Integer2 (2));

  public static final Integer2 z = (Integer2)
    (Integer2.add()).apply(((
    (new Record())
      .add("1", (x)))
      .add("2", (y))));

}
```

Fig. 2. Two records instantiated

SML Code:

```
val a = {name="John Doe", age=20}
val b = ("John Doe", 20)
```

Java Equivalent:

```
public static final Record a = (Record)
  ((new Record())
    .add("name", new String2("John Doe")))
    .add("age", (new Integer2 (20)));

 public static final Record b = (Record)
  ((new Record())
    .add("1", new String2("John Doe")))
    .add("2", (new Integer2 (20)));
```

model literally. An SML datatype, *dt*, with constructors *c1, c2, c3* ... is translated to a class. This class, also named *dt*, has static methods *c1, c2, c3* .... Each such method returns a new instance of *dt*.

Thus, SML code invoking a constructor becomes a static method call in the translated code. It is important to note that this process is different from the translation of normal SML functions. The special handling of type constructors greatly enhances translated code readability.

A datatype translation is given in figure 3. As the SML langauge enforces type safety, constructor arguments can simply be type *Object*. Although more restrictive types could be specified, there is little benefit in the common case where the type is *Record*.

2.4 Functions

In our translation model, the *Function* class encapsulates the concept of an SML function. Every SML function becomes an instance of this class. The Java *Function* class has a single method, *apply*, which takes an *Object* as its only parameter and returns an *Object*. The *Function* class encapsulation is necessitated by the fact that functions are treated as values in SML. As a byproduct of this scheme, function applications become intuitive; any application is translated to *Function_Name.apply(argument)*.

At an early design stage, the authors considered translating each function to a named class and a single instantiation of that class. While this model provides named functions that can be passed to other functions and otherwise treated as data, it does not easily accommodate anonymous functions. A strong argument for the current model is that instantiating anonymous subclasses of *Function* provides a natural way to deal with anonymous functions.

We believe this is a sufficiently general approach, and can handle all issues with respect to SML functions (including higher-order functions). In fact, every SML function declaration (i.e. named function) is translatead, by the SML/NJ compiler, to a recursive variable binding with an anonymous function. Therefore our treatment of anonymous functions and named functions mirror each other and this similarity lends itself to code readability.

Other authors have used different techinques for creating functions at runtime. For example, Kirby [7] uses the Java compiler to generate bytecode dynamically. While powerful and well suited for imperative programming, this approach is not compatible with the functional philosophy of SML.

In figure 4, the lines that contain the word “Pattern” form the foundation of what will, in future revisions of SML2Java, be fully generalized pattern matching. Pattern matching is done entirely at runtime, and consists of recursively comparing components of an expression’s value with a pattern. SML/NJ performs some optimizations of patterns at compile time [1]. However these optimizations are, in general, NP-hard [3] and SML2Java does not support them. Currently patterns are limited to records (including tuples), wildcards and integer constants.

Fig. 3. A dataype declaration and instantiation

SML Code:

```
datatype qux = FOO | BAR of int

val myVariable = FOO
val myOtherVar = BAR(42)
```

Java Equivalent:

```
public class TopLevel {
  public static class qux extends Datatype {

    protected qux(String constructor){
      super(constructor);
    }

    protected qux(String constructor,Object data){
      super(constructor, data);
    }

    public static qux BAR(Object o){
      return new qux("BAR", o);
    }

    public static qux FOO(){
      return new qux("FOO");
    }

  }
  public static final qux myVariable = (qux)
    qux.FOO();

  public static final qux myOtherVar = (qux)
    qux.BAR((new Integer2 (42)));

}
```

Fig. 4. Named function declaration and application

SML Code:

```
val getFirst = fn(x:int, y:int) => x
val one = getFirst(1,2)
```

Java Equivalent:

```
public static final Function getFirst = (Function)
  (new Function () {
    Object apply(final Object arg) {
      final Record rec = (Record) arg;
      RecordPattern pat = new RecordPattern();
      pat.add("1", new VariablePattern(new Integer2()));
      pat.add("2", new VariablePattern(new Integer2()));
      pat.match(rec);
      final Integer2 x = (Integer2) pat.get("1");
      final Integer2 y = (Integer2) pat.get("2");
      return (Integer2) (x);
    }
  });

public static final Integer2 one = (Integer2)
  (getFirst).apply(((
  (new Record())
    .add("1", (new Integer2 (1))))
    .add("2", (new Integer2 (2)))));
```

2.5 Let Expressions

A *Let* interface in Java encapsulates the SML concept of a let expression. The *Let* interface has no member functions. Every SML let expression becomes an anonymous instantiation of the *Let* interface with one member function, *in*. This function has no parameters and returns whatever type is appropriate given the original SML expression. The *in* function is called immediately following object instantiation.

A different approach would be to have the *Let* interface contain the function *in*. Here, *in* would have no formal parameters, and would return an *Object*. The advantage to this would be its consistency with respect to our function translations (i.e. the *apply* function), but a possible disadvantage is excessive typecasting, which can greatly reduce readability.

One might also attempt to separate the *Let* declaration from the call to its *in* function. If implemented in the most direct manner, such a model would, like the previous one, require that the *Let* interface contain an *in* function. This scheme would improve code readability. However, as one often has several *Let* expressions in the same name-space in SML, this model would likely suffer from shadowing issues.

Fig. 5. Let expressions are translated like functions

SML Code:

```
val x =
  let
    val y = 1
    val z = 2
  in
    y+z
  end
```

Java Equivalent:

```
public static final Integer2 x = (Integer2)
  (new Let() {
    Integer2 in() {
      final Integer2 y = (Integer2) (new Integer2 (1));
      final Integer2 z = (Integer2) (new Integer2 (2));
      return (Integer2) (Integer2.add()).apply(((
      (new Record())
        .add("1", (y)))
        .add("2", (z))));
    }
  }).in();
```

2.6 Module System

Our translation of SML's module system is straightforward. SML signatures are translated to abstract classes. SML structures are translated to classes that extend these abstract signature classes. A structure class only extends a given signature class if the original SML structure implements the SML signature. Structure declarations that are not externally visible in SML (i.e. not included in the implemented signature) are made private data-members in the generated Java structure class. This is demonstrated in figure 6.

3 Implementation

Our primary task was to translate high-level SML source code to high-level Java source code. As there are several available implementations of SML, we chose to use the front end of one, Standard ML of New Jersey (SML/NJ) [9]. We use the development flavor of the compiler (`sml-full-cm`) to parse and type-check input SML code. We then translate the abstract syntax tree generated by SML/NJ to our own internal Java syntax tree and output the Java code in source form.

Taking advantage of the SML/NJ type checker gives us a strong guarantee regarding the safety of the code we are translating. To cite Dr. Andrew Appel, a program produced from this code ”cannot corrupt the runtime system so that further execution of the program is not faithful to the language semantics” [2]. In other words, such a program cannot dump core, access private fields, or mistake types for one another. It would be interesting to investigate whether these facts, combined with the translation semantics of SML2Java, imply that similar guarantees hold in the generated Java code.

Other properties of the Core subset of SML are discussed by VanIngwegen [12]. Using HOL [5], she is able to prove, among other things, determinism of evaluation.

4 Conclusion and Future Goals

The current version of SML2Java translates many core constructs of SML, including primitive values, datatypes, anonymous and recursive functions, signatures and structures. SML2Java succeeds in translating SML to Java code, while respecting the functional paradigm.

Parametric polymorphism is a key construct that the authors would like to implement in SML2Java. Java 1.5 (due out late 2003) will directly support generics [11], and we believe waiting for Sun's implementation will facilate generating clean Java code. In addition, Java's generics will resemble C++ templates, and our treatment of parametric polymorphism should highlight the relative merits of each approach.

We would like to add support for several less critical SML constructs. Among these are exceptions, vectors, open declarations, mutual recursion, functors, and projects containing multiple files. The majority of these should be implementable without excessive difficulty, and each is expected to be a valuable addition to SML2Java.

Fig. 6. Translation of a signature and a structure

SML Code:

```
signature INDEX_CARD = sig
  val name : string
  val age : int
end

structure IndexCard :> INDEX_CARD = struct
  val name = "Professor Michael Jordan"
  val age = 31
  val super_secret = "This secret cannot be visible to the outside"
end
```

Java Equivalent:

```
public class TopLevel {
  private static abstract class INDEX_CARD  {
    public static final String2 name = null;
    public static final Integer2 age = null;
  }

  public static class IndexCard extends INDEX_CARD {
    public static final String2 name = (String2)
      (new String2 ("Professor Michael Jordan"));

    public static final Integer2 age = (Integer2)
      (new Integer2 (31));

    private static final String2 super_secret = (String2)
      (new String2 ("This secret cannot be visible to the outside"));

  }

}
```

5 Acknowledgements

This project was performed as independant research under the guidance of Dexter Kozen, Cornell University. We would like to thank Professor Kozen for many insightful discussions and much valuable advice. We would also like to thank the following for helpful advice: Andrew Myers, Cornell University, and Tore Larsen, Tromsø University.

Bibliography

[1] Aitken, William. *SML/NJ Match Compiler Notes* http://www.smlnj.org/compiler-notes/matchcomp.ps (1992)

[2] Appel, Andrew W. *A critique of Standard ML* http://ncstrl.cs.princeton.edu/expand.php?id=TR-364-92 (1992)

[3] Baudinet, Marianne and MacQueen, David. *Tree Pattern Matching for ML (extended abstract)* http://www.smlnj.org/compiler-notes/85-note-baudinet.ps (1985)

[4] Blume, Matthias. *No-Longer-Foreign: Teaching an ML compiler to speak C ”natively”* Electronic Notes in Theoretical Computer Science 59 No. 1 (2001)

[5] Gordon, Melham *Introduction to HOL. A theroem proving environment for higher order logic* Cambridge University Press, 1993

[6] Hicks, Michael. *Types and Intermdiate Representations* University of Pennsylvania (1998).

[7] Kirby, Graham, et al. *Linguistic Reflection in Java* Software - Practice & Experience 28, 10 (1998).

[8] Milner, Robin, et al. *The Definition of Standard ML - Revised.* Cumberland, RI: MIT Press, 1997.

[9] SML/NJ Fellowship, The. *Standard ML of New Jersey* http://www.smlnj.org (July 29, 2003).

[10] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.4.2 API Specification* http://java.sun.com/j2se/1.4.2/docs/api/ (July 18, 2003).

[11] Sun Microsystems. *JSR 14 Add Generic Types To The Java Programming Language* http://www.jcp.org/en/jsr/detail?id=14 (July 24, 2003).

[12] VanIngwegen, Myra. *Towards Type Preservation for Core SML* http://www.myra-simon.com/myra/papers/JAR.ps.gz

Constraint Imperative Programming with C++

Olaf Krzikalla

Reico GmbH krzikalla@gmx.de

Abstract. Constraint-based programming is of declarative nature. Problem solutions are obtained by specifying their desired properties, whereas in imperative programs the steps that lead to a solution must be defined explicitly. This paper introduces the Turtle Library, which combines constraint-based and imperative paradigms. The Turtle Library is based on the language Turtle[1] and enables constraint imperative programming with C++.

1 Constraint Imperative Programming at a Glance

In an imperative programming language the programmer describes how a solution for a given problem has to be computed. In contrast to that, in a declarative language the programmer specifies what has to be evaluated. Constraint-based programming is a rather new member of the declarative paradigm that was first developed from logic programming languages. In constraint-based programming the programmer describes the solution only by specifying the variables, their properties and the constraints over the set of variables. Actually, no algorithms have to be written. The compiler and runtime environment are responsible for providing appropriate algorithms and eventually obtaining a solution.

Meanwhile, constraint-based programming has been extended by concepts of other - mostly declarative - programming languages. However, the combination of imperative and constraint-based languages is far less explored. Borning and Freeman-Benson[2] introduced the term ’constraint-imperative programming’ and developed the language Kaleidoscope[3], combining constraint and object-oriented programming. But object-orientation is no precondition for constraint-imperative programming. This paper deals with more fundamental problems of the integration of constraints and constraint solvers in imperative language concepts. This integration promises some advantages. Imperative programming is a well known paradigm, which is intuitively understood by most programmers. A lot of efficient and industrial-strength imperative languages exist. However, an imperative program for a difficult algorithm is sometimes very cumbersome. Especially for this sort of problems declarative languages have proven their power. Constraint programming enables the programmer to specify required relations between objects directly rather than to ensure these relations by algorithms only. So constraint programs not only often become more compact and readable, but also less erroneous than their imperative counterparts.

Constraint imperative programming tries to combine the advantages of constraint-based and traditional imperative programming. A recent development in this field is the language Turtle, a constraint imperative programming language developed by Martin Grabmüller at the Technische Universität Berlin. Based on the ideas presented in [1] I developed the Turtle Library, a constraint imperative programming approach in C++.

2 The Basic Concept of Turtle

The fundamental difference between imperative and declarative languages is the model of time. In pure declarative languages a timing model simply does not exist - computations are specified independent of time. On the other hand, an imperative language always describes transformations of a given state at one point in time to another state at the next point in time. Computations are specified by sequences of statements.

Whenever declarative and imperative languages are combined, one of the main issues is the interaction of the integrated declarative concepts with the imperative timing model. In Turtle this is solved by introducing a lifetime for constraints and the statement *require*, which defines a constraint:

```
require constraint;
```

When a *require* is reached during the execution of the program, the given constraint is added to a global constraint store and taken into account during further computations - its lifetime starts. A constraint doesn't exist (and the system doesn't know anything about it) until the corresponding *require*-statement is executed. Eventually a sequence of *require*-statements form a conjunction of the appropriate constraints in the constraint store. Constraints in the constraint store are considered active.

Of course, if a constraint starts to exist at a certain time, it also can be removed at a certain time:

```
 require constraint in
   statement;
   ...
end;
```

The given constraint exists only between the *in* and *end*. When the program reaches the *end* statement (or otherwise leaves the block), the constraint is removed from the constraint store - its lifetime ends. After this the constraint isn't active any longer.

In order to deal with over- and underconstrained problems constraints need to be labelled with strenghts to form a constraint hierarchy. Altough a constraint imperative system without constraint hierarchies could be designed, its usefulness would be drastically reduced, because it would be difficult to constrain variables while the program dynamically adds or removes constraints. In Turtle each constraint can have a strength annotation in its definition:

```
require constraint1 : strong;
require constraint2 : mandatory;
```

When a constraint is annotated with a strength, it is added to the store with the given strength, otherwise with the strongest strength *mandatory*. This strength was specified in the previous example for clarity only.

Constraints are defined on constrainable variables. Most of the time a constrainable variable acts like a normal variable: it can be used in expressions and as a function argument. Only in a constraint statement they differ from their normal counterparts. A normal variable is treated like a constant, but a constrainable variable acts like a variable in the mathematical sense, and the constraint solver may change its value in order to satisfy all constraints existing at this point in time.

```
var x : int; // a normal variable
var y : ! int; // the exclamation defines a constrained variable
x := 0;
require y <= x in
  ... // during the execution of this block Turtle ensures y <= 0
end;
```

Constraints in Turtle are boolean expressions. During the execution of a *require* statement the constraint solver computes a certain value for each constrained variable, such that all active constraints evaluate to true. Constraints are handled strictly *eager*. Changing a non-constrained variable after it was used in a constraint doesn't affect the constraint store. Whenever the program reads a constrained variable, the value last computed by the solver for this variable is supplied. An exception is raised, if it isn't possible to satisfy all mandatory constraints during the execution of a *require* statement.

In Turtle constraints can be used for computing solutions to a certain problem like other constraint programming approaches. But they are not limited to this usage. *require* statements introduce conditions *a priori*, which are maintained automatically by the constraint solver. Hence backtracking like in approaches with *a posteriori* tests (e.g. Alma-0[6]) is not neccessary. Due to the *a priori* nature of constraints in Turtle they can be used to describe and preserve program invariants or - more general - to express in declarative manner the meaning of an otherwise imperative program without disrupting the familiar execution flow.

3 A Turtle in C++

The concepts of Turtle were first implemented in a language developed from scratch. This approach was chosen because some other features like higher-order functions should also be integrated. And a new language seemed to be the best choice for the seamless combination of imperative, functional and constraint programming. However, a new language is always in a difficult position. The knowledge base is small, tools don't exist, and further development is sometimes driven by academic interests only.

All concepts of Turtle related to constraint programming are also implementable in C++. Thats why I think a Turtle Library written in pure C++ serves both the widespreading and further development of Turtle better. In the recent years a lot of developments

- especially on the field of generic programming in C++ - made it possible to move almost all concepts from the Turtle language to the C++ Turtle Library without any losses. Furthermore, the generic approach of the Turtle Library enables every user to add, change or optimize constraint solvers at will. This is especially important for user-defined domains and offers a wide application field for the Turtle Library. The Turtle Library might be used to solve operational research problems or to program a graphical user interface. Both problems are typical constraint problems. In the first problem constraint programming is used only to obtain a solution, which often can be done in a constraint logic language too (given an appropriate language and - more important - an appropriate programmer) or by using a rather imperative approach[5]. But for the second problem constraint imperative programming really shines. The 'canonical' example is a graphical element, which can be dragged by the mouse inside certain borders[4]. The imperative approach looks like this:

```
void drag ()
{
  while (mouse.pressed) {  //message processing is left out
    int y = mouse.y;
    if (y > border.max)
      y = border.max;
    if (y < border.min)
      y = border.min;
    draw_element (fix_x, y, graphic);
  }
}
```

Using the Turtle Library the example would look as follows:

```
void drag ()
{
  constrained<int> y;
  require (y >= border.min && y <= border.max);
  while (mouse.pressed) {
    y = mouse.y;
    draw_element (fix_x, y(), graphic);
  }
}
```

The above is not only shorter, but expresses the relation between the border-object and the y-coordinate in exactly the way a programmer would think about it.

3.1 Constrained Variables

A constrained variable is of the generic type `constrained`. A constrained variable has identity semantics, the copy constructor and standard assignment operator aren't implemented. If they are needed, an appropriate wrapper (e.g. a reference counted pointer) has to be defined. The public interface given here is described in detail in the following sections.

```
template<class T>
class constrained
{
  public:
    constrained (const T& prefer = T());
    constrained<T>& operator= (const T& prefer);
    ˜constrained ();
    T operator ()() const throw (overconstrained_error, ...);
    const T& preferred() const;
    void unfix() const;
};
```

The template parameter specifies the value type of the variable. It might be a fundamental type like `int` or `double` or an user-defined class. Domains are formed by non-intersecting sets of value types and for each domain an appropriate constraint solver has to be provided. Thus each value type is unambiguously bound to a constraint solver. However Turtle can be used for hybrid domains, because the interface enables the implementation of a constraint solver responsible for more than one value type.

3.2 Declaring Constraints

Constraints can be declared as straightforward as presented in the section 2:

```
constrained<double> a, b;
double c = 2.0;
require (a >= 0.0);
require (a <= b && a + b <= c);
```

The composition of the boolean expression inside a `require` is done using operator overloading and expression template techniques. Which operators are supported for a certain value type is defined by the domain and the available constraint solver. E.g. it is rather pointless to support `>`, `<` or `!=` for floating point values[1]. In domains other than the algebraic ones it's often better to avoid otherwise meaningless operator overloading. For this purpose named predicates can be defined and used instead:

```
edge e = /*...*/;  //compute an edge
constrained<vertex> p;
require (point_on_edge (e, p));
```

The operator `&&` forms a conjunction of two expressions just like two subsequent requires, hence

```
constrained<double> a;
require (a >= 0 && a <= 2);
```

is equivalent to

[1] Due to the same reasons even the support of == could be argued.

```
constrained<double> a;
require (a >= 0);
require (a <= 2);
```

The operator `||` defines a disjunction. A disjunction can be seen as a branch in a tree of solutions. Subseqent requires add their constraints to all leafs of the tree.

```
constrained<double> a, b;
require (a == 0 || a == 1);
require (b == a + 1);
// the store now contains :
// (b == a + 1 && a == 0) || (b == a + 1 && a == 1)
```

The Turtle Library provides a simple generic algorithm for handling disjunctions. A certain constraint solver may implement a more sophisticated approach to compute and maintain solution trees efficiently.

Constraint strengths can be given as a second argument to require like in the Turtle language:

```
require (a == b, weak);
```

Of course these values are only of interest if the underlying constraint solver supports hierarchic constraints.

The Turtle Library internally stores the constraints in several constraint sub-stores. A constraint sub-store is defined as the set of all constraints over a set of constrained variables, where each variable of the set is linked to each other variable of the set. Two variables x and y are linked, if they either both appear in one constraint or if x appears in a constraint containing a variable linked to y.

```
constrained<double> a, b;
require (a >= 0.0);  // generate constraint sub-store 1
require (b >= 0.0);  // generate constraint sub-store 2
require (a <= b);    // sub-store 1 and 2 are merged
                     // together
```

The function template `require` returns a handle to manage the lifetime of the constraint. If the return value is ignored, the imposed constraint exists as long as all constrained variables in this constraint:

```
constrained<int> a;
{
  constrained<int> b;
  require (a == b);
  //...
//leaving the scope of b, hence a == b
//is removed from the constraint store:
}
```

Otherwise, the lifetime of the constraint is also bound to the lifetime of the returned constraint handle:

```
constrained<int> a, b;
{
  constraint_handle<int> z = require (a == b);
  //...
//leaving the scope of z, hence a == b
//is removed from the constraint store:
}
```

Still, the constraint exists no longer than all constrained variables in it. When the handle ceases to exist after the constraint did, it is ignored.

3.3 Obtaining Values from Constrained Variables

In a first version of the Turtle Lib, the `constrained<T>` class has an `operator T() const` member function to obtain the actual value of the constrained variable. However, it turns out that this operator sometimes conflicts with the generation of expression templates in a `require`. Thus the function call operator `operator()() const` was overloaded to read a value from a constrained variable:

```
std::cout << a();  //prints a value matching all constraints
                   // to a
```

Whenever this operator is invoked, the constraint solver is started to determine the value of the appropriate variable. How the value is determined depends mainly on the solver. When the store is overconstrained and no value can be determined, an exception of type `overconstrained_error` (derived from `std::logic_error`) is raised.

But more often underconstrained situations occur. For this purpose the Turtle Library supports a preferred value. A value of type T can be assigned to a `constrained<T>` or used to construct such a variable. This value then becomes the preferred value of the constrained variable. Now, if it turns out that more than one solution exists for a certain variable, the solution closest to the preferred value is taken:

```
constrained<double> a (3);
require (a <= 2.5);
std::cout << a();  // prints 2.5
```

To a certain degree the preferred value acts like a weak constraint. This is especially useful, if the constraint solver itself doesn't support constraint hierarchies. Thus a hierarchic constraint solver isn't as necessary as in the original Turtle language.

The evaluation of the preferred value is done by the solver implementation. It can be used to define a *threshhold* or *destination* value enabling the solver to terminate the search through the solution tree as soon as possible.

Some domains consist of incompareable values making it impossible to define a closest solution. In this case no general behaviour can be defined. Instead the solver implementation has to define the use of the preferred value.

3.4 Implicit Fixing

Once a value is determined for a constrained variable, this value has to be taken into account for further calculations. The constrained variable itself gets implicitly fixed to the determined value:

```
constrained<int> a (2), b (0);
require (a == b);
std::cout << a();  // prints 2
std::cout << b();  // also prints 2
```

Without implicit fixing the value of b would be evaluated to 0 and hence violate the required constraint a == b. Implicit fixing is done by generating a new constraint of the form variable == value. Due to this important side effect the evaluation order of constrained variables must be carefully considered. If the output lines of the above example were exchanged, both lines would print 0. And the following leads to unspecified behavior:

```
std::cout << a() << b(); // which variable is evaluated
                         // first?
```

The implicit fix is not immediatly added to the constraint sub-store but kept in a delay store inside the sub-store. If only one implicit fix exists in a constraint sub-store, and the same variable shall be evaluated again, the fix is erased before the evaluation (later in the process a new fix will be added). If more implicit fixes exist, always all are taken into account.

```
constrained<int> a (2), b (0);
require (a == b);
for (int i = 0; i < 3; ++i) {
  int j;
  std::cin >> j;
  a = j;
  // prints j, because the only implicit fixed variable is a:
  std::cout << a();
}

constrained<int> a (2), b (0);
require (a == b);
std::cout << b();  // prints 0, fixes b
for (int i = 0; i < 3; ++i) {
  int j;
  std::cin >> j;
  a = j;
  // always prints 0, because b is fixed, but a is evaluated:
  std::cout << a();
}
```

As shown in the last example, sometimes implicit fixes are harmful, especially if more than one variable is evaluated inside a loop. That's why a constrained variable can be unfixed explicitly via the member function `unfix()`:

```
constrained<int> a (2), b (0);
require (a == b);
for (int i = 0; i < 3; ++i) {
  int j;
  std::cin >> j;
  a = j;
  std::cout << a();  // prints j and get fixed
  std::cout << b();  // prints also j and get fixed
  // now more than one fix exist, so all fixes would be
  // considered during further evaluations unless we
  // explicitly unfix the variables:
  a.unfix();
  b.unfix();
}
```

The computation of a value for a constrained variable differs a lot from the original Turtle language. While in the Turtle language the values of constrained variables are already determined during a require statement, the Turtle Library delays the computation until a read-action to a constrained variable occurs. The disadvantage of this approach seems the need of implicit fixing, which isn't part of the Turtle language[2].

On the other hand the delay of the computation offers some advantages. First, only when the computation is delayed until a read-action, the preferred value can be evaluated correctly. Otherwise a change of the preferred value after some requires could be ignored. Second, a solver knows which constrained variable actually is being read, can consider this fact during the computation and hence doesn't have to evaluate all variables in every case. And third, lazy evaluation becomes possible. Altough also the Turtle Library handle constraints *eager* mostly, it is not limited to this.

3.5 Lazy Evaluation

Lazy evaluation is an often arising issue when declarative and imperative concepts are combined. Shall a subexpression in a require-statement be evaluated immediately or shall the evaluation be delayed until the constraint is actually needed for the evaluation of a constrained variable? Consider the example:

```
int foo();

int example()
{
  int_c a, b
  int i = 1;
```

[2] Altough there is an ongoing argument about this topic.

```
  require (a == i);
  require (b >= foo());
  require (a < b);
  i = 2;
  std::cout << a();  // 1 or 2 ?, is foo() called here ?
  std::cout << b();  // or is foo() called only here ?
}
```

As stated earlier the Turtle Library doesn't perform lazy evaluation by default. This decision was made mainly due to lifetime issues. In C++ it's impossible to ensure that an arbitrary object exists until all constraints referring to it are erased. Hence the above example prints 1 for `a` and calls `foo()` during the evaluation of the argument for the second `require`. This has the additional benefit, that possible side effects of functions inside constraints are more predictable. If `foo()` would be lazy evaluated in the example above, it could be called once or twice, depending on the actual implementation of the underlying constraint solver.

Lazy evaluation can be simulated through the lifetime management of constraints. But sometimes it is just better to have some lazy evaluated values. Therefore a simple lazy evaluated value type is provided by the Turtle Library:

```
template<class T>
class lazy_evaluated
{
  public:
    explicit lazy_evaluated (const T& init = T());
    operator T() const;
    operator T&();
};
```

This class mostly acts like a value of type T, but its actual value is garbage collected (the copy constructor and assignment operator of `lazy_evaluated<T>` has identity semantics). Each constraint using a lazy evaluated variable stores a copy of the corresponding `lazy_evaluated<T>` variable. The actual value is preserved unless all references to it are removed. It is only read by the constraint solver when needed during the evaluation of a constrained variable. Side effects may only happen due to the copying of T.

```
int_c a;
lazy_evaluated<int> i = 1;
require (a == i);
i = 2;
std::cout << a();  //reads i at this point and thus prints 2
```

4 Programming with the Turtle Library

The Turtle Library can be downloaded from
`http://home.t-online.de/home/krize6/turtle.htm`.

At this page also some technical issues are discussed in more detail. Especially the steps needed to integrate a new constraint solver in the Turtle Library are described. Furthermore some more sophisticated examples of constraint imperative programming are already provided. They demonstrate the use of some techniques and little patterns to make constraint imperative programming more convenient and flexible.

4.1 User-defined Constraints and Dynamic Expressions

Often the declarative power of expression templates is sufficient to express the constraints in a compact and readable manner. But some constraints are so common that they deserve an own name. Such user-defined constraints can be generated using the function template `build_constraint`, which takes an constraint just like require, but only builds the internal representation of the given expression without adding it to the constraint store.

```
typedef constrained<int> int_c;

constraint_solver<int>::expr domain (const int_c& x,
                                     int min,
                                     int max)
{
  return build_constraint (x >= min && x <= max);
}

int_c a, b, c;
require (domain (a, 0, 9));
require (domain (b, 0, 99));
require (domain (c, -1, 1));
```

The naming of complex static expressions further enhances the readability of a program. But besides this constraint imperative programming also needs a way to create constraints dynamically. For this the Turtle Library provides a generic class `dynamic_expr`, which holds an (sub)expression and can be used like that, but has value semantics. A rather complex example is the function `example_dynamic_puzzle`, which is part of the sample file provided on the internet page of the Turtle Library.

4.2 Optimization

Constraint programming supplies a lot of tools to optimize a given function for a given set of constraints. Optimization is one the main usages of constraint programming. Hence, optimization should be possible with the Turtle Library, too. By using a preferred value for a given expression, optimization can be done without the needs of special library functions. Consider the following example:

```
double_c x, y;
require (y >= 0);
require (y >= 3 - 2 * x);
```

Given these constraints the sum of x and y shall be minimized. These can be done by a little pattern of the following three lines:

```
double_c min (- 1000.0);
require (min == x + y);
std::cout << min(); // prints 1.5
```

First a constrained variable has to be declared and the preferred value have to be set to an absolute minimal or maximal border [3]. Second, this variable has to be set equal to the expression to be optimized. And third, by reading the variable the value closest to the given preferred value gets calculated and stored in the variable. Furthermore the implicit fixing also immediately limits other constrained variables to values at the searched optimum.

5 Conclusion and Future Works

The Turtle Library defines an interface for the integration of constraint programming concepts in an imperative language and provides an implementation of this interface for a popular language. Hopes are, that this opens a wider application field for constraint imperative programming. Only the practical use will show further needs. E.g. if an implicit fix of a constrained variable has to be considered is defined by a rather complex rule. It's unclear if this rule is of any practical value. Also, for the moment there is no way to unfix a bunch of variables at once (e.g. all variables of a sub-store).

The class `lazy_evaluated<T>` should be treated as a simple example for lazy evaluation. It is possible to further parametrize this class to allow more complex actions during constraint evaluation including the call of functions. If this is done, side effects of a lazy evaluated function has to be considered carefully as stated in section 3.5. At the moment it's quite unclear if the gain of flexibility outweighs the possiblity of near unpredictable side effects.

The modelling of algebraic problems using the Turtle Library is already very convenient. But the generic approach offers a lot more. A lot of publications in the recent decade has shown, that constraint programming is well-suited for several problem domains. But unfortunately a lot of these publications either introduced a whole new language or at least extended an existing language by adding new language constructs (and thus became incompatible to the parent language). But an application programmer can't just move from one language to the next at will. Due to business, management and also educational issues he has to stick to one - often for years. With the Turtle Library now even the application programmer gets a tool to use constraints in C++ in the convenient declarative manner as it is already used for years in other languages.

[3] This example is rather abstract and hence knows no 'absolute' minimum. In practical applications it should be always possible to find a reasonable value (see also `example_knapsack`).

Bibliography

[1] Grabmüller, M.: Constraint Imperative Programming. Diploma Thesis, Technische Universität Berlin 2003,

[2] Freeman-Benson, B.N.: Constraint Imperative Programming. PhD Thesis, University of Washington, 1991. Published as Department of Computer Science and Engenieering Technical Report 91-07-02

[3] Borning, A. and Freeman-Benson, B.N.: The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *Proceedings of the IEEE Computer Society 1992 International Conference on Computer Languages*, pages 174-180, 1992

[4] Lopez, G.: The design and implementation of Kaleidoscope, a constraint imperative programming language. PhD Thesis, University of Washington, 1997.

[5] ILOG. ILog Web Site.
`http://www.ilog.com`, last visited 2003-06-23

[6] Apt, K.R., Brunekreef, J., Partington, V. and Schaerf, A.: Alma-0: An imperative language that supports declarative programming. ACM Toplas, 20(5):1014-1066, 1998.

Patterns in Datatype-Generic Programming

Jeremy Gibbons

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
jeremy.gibbons@comlab.ox.ac.uk

Abstract. *Generic programming* consists of increasing the expressiveness of programs by allowing a wider variety of kinds of parameter than is usual. The most popular instance of this scheme is the C++ Standard Template Library. *Datatype-generic programming* is another instance, in which the parameters take the form of datatypes. We argue that datatype-generic programming is sufficient to express essentially all the genericity found in the *Standard Template Library*, and to capture the abstractions motivating many *design patterns*. Moreover, datatype-generic programming is a precisely-defined notion with a rigorous mathematical foundation, in contrast to generic programming in general and the C++ template mechanism in particular, and thereby offers the prospect of better static checking and a greater ability to reason about generic programs. This paper describes work in progress.

1 Introduction

Generic programming [28, 19] is a matter of making programs more adaptable by making them more general. In particular, it consists of allowing a wider variety of entities as parameters than is available in more traditional programming languages.

The most popular instantiation of generic programming today is through the C++ Standard Template Library (STL). The STL is basically a collection of container classes and generic algorithms operating over those classes. The STL is, as the name suggests, implemented in terms of C++'s template mechanism, and thereby lies both its flexibility and its intractability.

Datatype-generic programming (DGP) is another instantiation of the idea of generic programming. DGP allows programs to be parameterized by a *datatype* or *type functor*. DGP stands and builds on the formal foundations of category theory and the *Algebra of Programming* movement [8, 7, 10], and the language technology of Generic Haskell [22, 12].

In this paper, we argue that DGP is sufficient to express essentially all the genericity found in the STL. In particular, we claim that various programming idioms that can at present only be expressed informally as *design patterns* [17] could be captured formally as datatype-generic programs. Moreover, because DGP is a precisely-defined notion with a rigorous mathematical foundation, in contrast to generic programming in general and the C++ template mechanism in particular, this observation offers the prospect of better static checking of and a greater ability to reason about generic programs than is possible with other approaches.

This paper describes work in progress — in fact, it describes work largely in the future. The United Kingdom's Engineering and Physical Sciences Research Council is funding a project called *Datatype Generic Programming*, starting around September 2003. The work described in this paper will constitute about a third of that project; a second strand, coordinated by Roland Backhouse at Nottingham, is looking at more of the underlying theory, including logical relations for modular specifications, higher-order naturality properties, and termination through well-foundedness; the remainder of the project consists of an integrative case study.

The rest of this paper is structured as follows. Section 2 describes the principles underlying the C++ Standard Template Library. Section 3 motivates and defines Datatype-Generic Programming, and explains how it differs from a number of similar approaches to genericity. Section 4 discusses the Design Patterns movement, and presents our case for the superiority of datatype genericity over informal prose for capturing patterns. Section 5 concludes by outlining our future plans for the DGP project.

2 Principles Underlying the STL

The STL [6] is structured around four underlying notions: *container types*, *iterators*, *algorithms*, and *function objects*. These notions are grouped into a hierarchy (in fact, a directed acyclic graph) of *concepts*, representing different abstractions and their relationships. The library is implemented using the C++ *template mechanism*, which is the only means of writing generic programs in C++. This section briefly analyzes these six principles, from a functional programmer's point of view.

2.1 The C++ Template Mechanism

The C++ template mechanism provides a means for classes and functions to be parametrized by types and (integral, enumerated or pointer) values. This allows the programmer to express certain kinds of abstraction that otherwise would not be available. A typical example of a function parametrized by a type is the function *swap* below:

```
template⟨class T⟩
void swap(T& a, T& b) { T c = a; a = b; b = c; }

main() {
  int i1 = 3, i2 = 4;             swap⟨int⟩(i1, i2);
  double d1 = 3.5, d2 = 4.5;      swap⟨double⟩(d1, d2);
}
```

The same function template is instantiated at two different types to yield two different functions. Container classes form typical examples of parametrization of a class by a type; the example below shows the outline of a *Vector* class parametrized by size and by element type.

```
template⟨class T, int size⟩
class Vector {private : T values[size]; ...};

main() {
   Vector⟨int, 3⟩ v;
   Vector⟨Vector⟨double, 100⟩, 100⟩ matrix;
}
```

The same class template is instantiated three times, to yield a one-dimensional vector of three integers and a two-dimensional 100-by-100 matrix of doubles.

A template is to all intents and purposes a macro; little is or can be done with it until the parameters are instantiated, but the instantiations that this yields are normal code and can be checked, compiled and optimized in the usual way. In fact, the decision about which template instantiations are necessary can only be made when the complete program is available, namely at link time, and typically the linker has to call the compiler to generate the necessary instantiations.

The C++ template mechanism is really a *special-purpose, meta-programming* technique, rather than a general-purpose generic-programming technique. Meta-programming consists of writing programs in one language that generate or otherwise manipulate programs written in another language. The C++ template mechanism is a matter of meta-programming rather than programming because templated code is not actually 'real code' at all: it cannot be type-checked, compiled, or otherwise manipulated until the template parameter is instantiated. Some errors in templated code, such as syntax errors, can be caught before instantiation, but they are in the minority; static checking of templates is essentially impossible. Thus, a class template is not a formal construct with its own semantics — it is one of the ingredients from which such a formal entity can be constructed, but until the remaining ingredients are provided it is merely a textual macro. In a programming language that offers such a template mechanism as its only support for generic programming, there is no hope for a calculus of generic programs: at best there can be a calculus of their specific instances.

The template mechanism is a special-purpose, as opposed to general-purpose, meta-programming technique, because only limited kinds of compile-time computation can be performed. Actually, the mechanism provides surprising expressive power: Unruh [38] demonstrated the disquieting possibility of a program whose compilation yields the prime numbers as error messages, Czarnecki and Eisenecker [13] show the Turing-completeness of the template mechanism by implementing a rudimentary LISP interpreter as a template meta-program, and Alexandrescu [4] presents a tour-de-force of unexpected applications of templates. But even if technically template meta-programming has great expressiveness, it is pragmatically not a convenient tool for generating programs; applications of the technique feel like tricks rather than general principles. Everything computable is expressible, albeit sometimes in unnatural ways. A true general-purpose meta-programming language would support 'programs as data' as first-class citizens, and simple and obvious (as opposed to 'surprising') techniques for manipulating such programs [35].

There are several consequences of the fact that templated code is a meta-program rather than (a fragment of) a pure program. They all boil down to the fact that separate compilation of the templated code is essentially impossible; it isn't real code until it is instantiated. Therefore:

- templated code must be distributed in source rather than binary form, which might be undesirable (for example, for intellectual property reasons);
- static error checking is in general precluded, and any errors are revealed only at instantiation time; moreover, error reports are typically verbose and unhelpful, because they relate to the consequences of a misuse rather than the misuse itself;
- there is a problem of 'code bloat', because different instantiations of the same templated code yield different units of binary code.

There is work being done to circumvent these problems by resorting to partial evaluation [39], but there is no immediate sign of a full resolution.

2.2 Container Types

A *container type* is a type of data structures whose purpose is to contain elements of another type, and to provide access to those elements. Examples include arrays, sequences, sets, associative mappings, and so on.

To a functional programmer, this looks like a *polymorphic datatype*; for example,

$$\mathbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

A data structure of type $\mathit{List}\ \alpha$ for some α will indeed contain elements of type α, and will (through pattern-matching, for example) provide access to them. Such polymorphic datatypes can be given a formal semantics via the categorical notion of a *functor* [10], an operation simultaneously on types (taking a type α to the type $\mathit{List}\ \alpha$) and functions (taking a function of type $\alpha \to \beta$ to the map function of type $\mathit{List}\ \alpha \to \mathit{List}\ \beta$).

However, that response is a little too simple. Certainly, some polymorphic datatypes and some functors correspond to container types, but not all do. For example, consider the polymorphic type

$$\mathbf{data}\ \mathit{Transformer}\ \alpha = \mathit{Trans}\ (\alpha \to \alpha)$$

(The natural way to define this type in Haskell [34] is with a type synonym rather than a datatype declaration, but we've chosen the latter to make the point clearer.) There is no obvious sense in which a data structure of type $\mathit{Transformer}\ \alpha$ 'contains' elements of type α. Hoogendijk and de Moor [24] have shown that one wants to restrict attention to the functors with a *membership* operation. Technically, in their relational setting, the membership of a functor F is the largest lax natural transformation from F to Id, the identity functor; informally, membership is a non-deterministic mapping selecting an arbitrary element from a container data structure. Some functors, such as $\mathit{Transformer}$, have no membership operation, and so do not correspond to container types according to this definition.

2.3 Iterators

The essence of the STL is the notion of an *iterator*, which is essentially an abstraction of a pointer. The elements of a container data structure are made accessible by providing iterators over them; the container typically provides operations $begin()$ and $end()$ to yield pointers to the first element and to 'one step beyond' the last element.

Basic iterators may be compared for equality, dereferenced and incremented. But there are many different varieties of iterator: *input iterators* may be dereferenced only as R-values (for reading), and *output iterators* only as L-values (for writing); *forward iterators* may be deferenced in both ways, and may also be copied (so that multiple elements of a data structure may be accessed at once); *bidirectional iterators* may also be decremented; and *random-access iterators* allow amortized constant-time access to arbitrary elements.

Despite the name, iterators in the STL do not express exactly the same idea as the ITERATOR design pattern, although they have the same intent of 'providing a way to access the elements of an aggregate object sequentially without exposing its underlying representation' [17]. In fact, the proposed design in [17] is fairly close to an STL input iterator: an existing collection may be traversed from beginning to end, but the identities of the elements in the collection cannot be changed (although their state may be).

What all these varieties of iterator have in common, though, is that they point to individual elements of the data structure. This is inevitable given an imperative paradigm: as Austern [6] puts it, 'The moving finger writes, and having writ, moves on', and although under more refined iterator abstractions the moving finger may rewrite, and may move backwards as well as forwards, it is still a finger pointing at a single element of the data structure.

One functional analogue of iterators for traversing a data structure is the map operator that arises as the functorial action on element functions, acting on each element independently. More generally, one could point to *monadic maps* [15], which act on the elements one by one, using the monad to thread some 'state' through the computation.

However, lazy functional programmers are liberated by the availability of 'new kinds of glue' [26] for composing units of code, and have other options too. For example, they may use lists to achieve a similar separation of concerns: the interface between a collection data structure and its elements is via a list of these elements. The analogue to the distinction between input and output iterators (R-values and L-values) is the provision of one function to yield the *contents* of a data structure as a list of elements, and another to *generate* a new data structure from a given list of elements.

This functional insight reveals a rather serious omission in the STL approach, namely that it only allows the programmer to manipulate a data structure in terms of its elements. This is a very small window through which to view the data structure itself. A map ignores the shape of a data structure, manipulating the elements but leaving the shape unchanged; iterator-style access also (deliberately) ignores the shape, flattening it to a list. Neither is adequate for capturing problems that exploit the shape of the data, such as pretty-printers, structure editors, transformation engines and so on. A more general framework is obtained by providing *folds* to consume data structures and *unfolds* to generate them [18] — indeed, the $contents$ and $generate$ functions mentioned above are instances of folds and unfolds respectively, and a map is both a fold and an unfold.

2.4 Concepts

We noted in the previous section that the essence of the STL is a hierarchy of varieties of iterator. In the STL, the members of this hierarchy are called *concepts*. Roughly speaking, a concept is a set of requirements on a type (in terms of the operations that are available, the laws they satisfy, and the asymptotic complexities in time and space); equivalently, a concept can be thought of as the set of all types satisfying those requirements.

Concepts are not part of C++; they are merely an artifact of the STL. An STL reference manual [6] can do no more than to describe a concept in prose. Consequently, it is a matter of informal argument rather than formal reasoning whether a given type is or is not a model of a particular concept. This is a problem for users of the STL, because it is easy to make mistakes by using an inappropriate type in a particular context: the compiler cannot in general check the validity of a particular use, and tracking down errors can be tricky. There have been some valiant attempts to address this problem by programming idioms [36, 31] or static analysis [21], but ultimately the language seems to be a part of the problem here rather than a part of the solution.

The solution seems obvious to the Haskell programmer: use type classes [29]. A type class captures a set of requirements on a type, or equivalently it describes the set of types that satisfy those requirements. (Type classes are more than just interfaces: they can provide default implementations of operations too, and type class inference amounts to automatic selection of an implementation.) Type classes are only an approximation to the notion of a concept in the STL sense, because they can capture only the signatures of operations and not their extensional (laws) or intensional (complexity) semantics. However, they are statically checkable within the language, which is at least a step forwards: C++ concepts cannot even capture signatures formally. The Haskell collection class library Edison [11, 33] uses type classes formally in the same way that STL uses concepts informally.

2.5 Algorithms and Function Objects

The bulk of the STL, and indeed its whole raison d'être, is the family of generic *algorithms* over container types made possible by the notion of an iterator. These algorithms are general-purpose operations such as searching, sorting, comparing, copying, permuting, and so on. Iterators decouple the algorithms from the container types on which they operate: the algorithm is described in terms of an abstract iterator interface, and is then applicable to any container type on which an appropriate iterator is available.

There is no new insight provided by the algorithms per se; they arise as a natural consequence of the abstractions provided (whether informally as concepts or formally as type classes) to access the elements of container types. In the STL, algorithms are represented as function templates, parametrized by models of the appropriate iterator concept. To a Haskell programmer, algorithms in this sense correspond to functions with types qualified by a type class.

The remaining principle on which the STL is built is that of a *function object* (sometimes called a 'functor', but in a different sense that the functors of category theory). Function objects are used to encapsulate function parameters to algorithms; typical uses are for parametrizing a search function by a predicate indicating what to search for, or a sorting procedure by an ordering.

Function objects also yield no new insight to the functional programmer. In the STL, a function object is represented as an object with a single method which performs the function. This is essentially an instance of the STRATEGY design pattern [17]. To a functional programmer, of course, function objects are unnecessary: functions are first-class citizens of the language, and a function can be passed as a parameter directly.

3 Datatype Genericity

We propose a new paradigm for generic programming, which we have called *datatype-generic programming* (DGP). The essence of DGP is the parametrization of values (for example, of functions) by a *datatype*. We use the term 'datatype' here in the sense discussed in Section 2.2: a container type, or more formally a functor with a membership operation. For example, 'List' is a datatype, whereas 'int' is merely a type.

(Since a datatype is one type parametrized by another — 'lists of αs, for some type α' — and a datatype-generic program is a program parametrized in turn by such a type-parametrized type, we toyed briefly with the idea of describing our proposal as for a *'type-parametrized–type'—parametrized theory of programming*, or TPTPTP for short. But we decided that was a bit of a mouthful.)

3.1 An Example of DGP

Consider for example the parametrically polymorphic programs $\mathit{maplist}$,

$$\begin{array}{l}
\mathit{maplist} :: (\alpha \to \beta) \to \mathit{List}\ \alpha \to \mathit{List}\ \beta \\
\mathit{maplist}\ f\ \mathit{Nil} \qquad\qquad\ = \mathit{Nil} \\
\mathit{maplist}\ f\ (\mathit{Cons}\ a\ x) = \mathit{Cons}\ (f\ a)\ (\mathit{maplist}\ f\ x)
\end{array}$$

and (for the appropriate definition of the Tree datatype) $\mathit{maptree}$,

$$\begin{array}{l}
\mathit{maptree} :: (\alpha \to \beta) \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \beta \\
\mathit{maptree}\ f\ (\mathit{Tip}\ a) \quad\ = \mathit{Tip}\ (f\ a) \\
\mathit{maptree}\ f\ (\mathit{Bin}\ x\ y) = \mathit{Bin}\ (\mathit{maptree}\ f\ x)\ (\mathit{maptree}\ f\ y)
\end{array}$$

Both of these programs are already quite generic, in the sense that a single piece of code captures many different specific instances. However, the two programs are themselves clearly related, and a DGP language would allow their common features to be captured in a single definition map:

$$\begin{array}{ll}
\mathit{map}\langle \mathit{Unit} \rangle\ () & = () \\
\mathit{map}\langle \mathit{Const}\ a \rangle\ x & = x \\
\mathit{map}\langle + \rangle\ f\ g\ (\mathit{Inl}\ u) & = \mathit{Inl}\ (f\ u) \\
\mathit{map}\langle + \rangle\ f\ g\ (\mathit{Inr}\ v) & = \mathit{Inr}\ (g\ v) \\
\mathit{map}\langle \times \rangle\ f\ g\ (u, v) & = (f\ u, g\ v)
\end{array}$$

This single definition is parametrized by a datatype; in this case it is defined by structural induction over a grammar of datatypes. The two parametrically polymorphic programs are of course instances of this one datatype-generic program: $maplist = map\langle List\rangle$ and $maptree = map\langle Tree\rangle$.

At first glance, this looks rather like a generic algorithm that could have come from the STL, and indeed in this case that is a valid analogy to make: map-like operations can be expressed in the STL. However, the crucial difference is that DGP allows a program to *exploit* the shape of the data on which it operates. For example, one could write datatype-generic functions to encode a data structure as a bit string and to decode the bit string to regenerate the data structure [27]: the *shape* of the data structure is related to the *value* of the bitstring. A more sophisticated example involves Huet's 'Zipper' [25] for efficiently but purely functionally representing a tree with a cursor position; different types of tree require different types of zipper, and it is possible [1, 23] to write datatype-generic operations on the zipper: here, the shape of one data structure determines the shape of an auxilliary data structure in a rather complicated fashion. Neither of these examples are possible with the STL.

3.2 Isn't This Just. . . ?

As argued above, the parametrization of programs by datatypes is not the same as *generic programming* in the STL sense. The latter allows *abstraction from* the shape of data, but not *exploitation of* the shape of data. Indeed, this is why we chose a new term 'DGP' instead of simply using 'GP': we would prefer the latter term, but feel that it has already been appropriated for a more specific use than we would like. (For example, one often sees definitions such as 'Generic programming is a methodology for program design and implementation that separates data structures and algorithms through the use of abstract requirement specifications' [37, p19]. We feel that such definitions reduce generic programming to good old-fashioned abstraction.)

DGP is not the same thing as *meta-programming* in general, and template meta-programming in particular. Meta-programming is a matter of writing programs that generate or otherwise manipulate other programs. For example, C++ template meta-programs yield ordinary C++ code when instantiated (at least notionally, although the code so generated is typically never seen); they are not ordinary C++ programs in their own right. A meta-program for a given programming language is typically not a program written in that language, but one written in a meta-language that generates the object program when instantiated or executed. In contrast, a datatype-generic program is a program in its own right, written in (perhaps an enrichment of) the language of the object program.

Neither is DGP the same thing as polymorphism, in any technical sense we know. It is clearly not the same thing as ordinary *parametric polymorphism*, which allows one to write a single program that can manipulate both lists of integers and lists of characters, but does not allow one to write a single program that manipulates both lists of integers and trees of integers. We also believe (but have yet to study this in depth) that DGP is not the same thing as *higher-order parametric polymorphism* either, because in general the programs are not parametric in the functor parameter: if they were, they might manipulate the shape of data but could not compute with it, as with the encoding and decoding example cited above.

Nor is it the same thing as *dependently typed programming* [5], which is a matter of parametrizing types by values rather than values by types. Dependent types are very general and powerful, because they allow the types of values in the program to depend on other values computed by that program; but by the same token they rule out the possibility of most static checking. (A class template parametrized by a value rather than a type bears some resemblance to type dependent on a value, but in C++ the actual template parameters must be statically determined for instantiation at compile time, whereas dependent type theory requires no such separation of stages.) It would be interesting to try to develop a calculus of dependently typed programming, but that is a different project altogether, and a much harder one too.

Finally, DGP is not simply Generic Haskell [12], although the datatype-generic program for map we showed above is essentially a Generic Haskell program. The Generic Haskell project is concentrating on the design and implementation of a language that supports DGP, but is not directly addressing the problem of developing a calculus of such programs. Our project has strong connections with the Generic Haskell project, and we are looking forward to making contributions to the design based on our theory-driven insights, as the language is making contributions to the theory by posing the question of how it may be used. However, Generic Haskell is just one possible implementation technique for DGP.

4 Patterns of Software

A *design pattern* ‘systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems’ [17]. The intention is to capture best practice and experience in software design in order to facilitate the education of novices in what constitutes good designs, and the communication between experts about those good designs. The software patterns movement is based on the work of Christopher Alexander, who for over thirty years has been leading a similar movement in architecture [3, 2].

It could be argued that many of the patterns in [17] are idioms for mimicking DGP in languages that do not properly support such a feature. Because of the lack of proper language support, a pattern can generally do no better than to motivate, describe and exemplify an idiom: it can refer indirectly to the idiom, but not present the idiom directly as a formal construction. For example, the ITERATOR pattern shows how an algorithm that traverses the elements of a collection type can be decoupled from the collection itself, and so can work with new and unforeseen collection types; but for each such collection type an appropriate new ITERATOR class must be written. (The programmer may be assisted by the library, as in Java [20], or the language, as in C♯ [14], but still has to write something for each new collection type.) A language that supported DGP would allow the expression of a single datatype-generic program directly applicable to an arbitrary collection type: perhaps a function to yield the elements as a lazy list, or a map operation to transform each element of a collection.

The situation is no better with the STL than with design patterns. We argued above that iterators in the STL sense are more general than the ITERATOR pattern. Nevertheless, C++ provides no support for defining the iterator concept, so it too can only be referred to indirectly; and again, for every new collection type an appropriate implementation of the concept must be provided.

As another example, the VISITOR pattern [17] allows one to decouple a multivariant datatype (such as abstract syntax trees for a programming language) from the specific traversals to be performed over that datatype (such as type checking, pretty printing, and so on), allowing new traversals to be added without modifying and recompiling each of the datatype variants. However, each new datatype entails a new class of VISITOR, implemented according to the pattern. A DGP language would allow one to write a single datatype-generic traversal operator (such as a *fold*) once and for all multivariant datatypes.

(Alexandrescu [4] does present a 'nearly generic' definition of the VISITOR pattern using clever template meta-programming, but it relies on C++ macros, and still requires the foresight in designing the class hierarchy to insert a call to this macro in every class in the hierarchy that might be visited.)

It is sometimes said that patterns cannot be automated; anything that can be captured completely formally is too restricted to be a proper pattern. Alexander describes a pattern as giving 'the core of the solution to [a] problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice' [3]; Gamma et al. state that 'design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is' [17]. We are sympathetic to the desire to ensure that patternity does not become a synonym for 'a good idea', but do not feel that that means we should give up on attempts to formalize patterns.

Alexander, in his foreword to Gabriel's book [16], hopes that the software patterns movement will yield 'programs which make you gasp because of their beauty'. We think that's a goal worth aiming for, however optimistically. We have yet to see a meta-programming framework that supports beautiful programming (although we confess to being impressed by the intricate possibilities of template meta-programming demonstrated by [4]), but we have high hopes that datatype-generic programs could be breathtakingly beautiful.

5 Future Plans

The DGP project is due to start around September 2003; the work outlined in this paper constitutes about a third of the total. One of the initial aims of this strand will be an investigation into the relationships between generic programming (as exhibited in libraries like the STL), structural and behavioural design patterns (as described by [17]), and the mathematics of program construction (epitomized by Hoogendijk and de Moor's categorical characterization of datatypes [24]).

In the short term, we intend to use the insights gained from this investigation to prototype a datatype-generic collection library in Generic Haskell [12] (perhaps as a refinement of Okasaki's Edison library [33]). This will allow us to replace type-unsafe meta-programming with type-safe and statically checkable datatype-generic programming. Ultimately, however, we hope to be able to apply these insights to programming in more traditional object-oriented languages, perhaps by compilation from a dedicated DGP language.

But the real purpose of the project will be to generalize theories of program calculation such as Bird and de Moor's relational 'algebra of programming' [10], to make it more applicable to deriving the kinds of programs that users of the STL write. This will link with Backhouse's strand of the DGP project, which is looking at more theoretical aspects of datatype genericity: higher-order naturality properties, logical relations, and so on. We intend to build on this work to develop a calculus for generic programming.

More tangentially, we have been intrigued by similarities between some of the more esoteric techniques for template meta-programming [13, 4] and some surprising possibilities for computing with type classes in Haskell [32, 30, 9]. It isn't clear yet whether those similarities are a coincidence or evidence of some deeper correspondence; in the light of our arguments in this paper that type classes are the Haskell analogue of STL concepts, we suspect there may be some deep connection here.

6 Acknowledgements

The help of the following people and organizations is gratefully acknowledged:

- Roland Backhouse, Graham Hutton, Ralf Hinze and Johan Jeuring, for their contributions to the DGP grant proposal;
- Richard Bird, for inspiring and encouraging this line of enquiry;
- Tim Sheard, for his elegant definition of generic programming;
- EPSRC grant GR/S27078/01, for financial support.

Bibliography

[1] Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In Martin Hofmann, editor, *LNCS 2701: Typed Lambda Calculi and Applications*, pages 16–30. Springer-Verlag, 2003.

[2] Christopher Alexander. *The Nature of Order*. Oxford University Press, To appear in 2003.

[3] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.

[4] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[5] Lennart Augustsson. Cayenne: A language with dependent types. *SIGPLAN Notices*, 34(1):239–250, 1999.

[6] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.

[7] R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In M. Nivat, C.S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91*, pages 303–326. Springer-Verlag, Workshops in Computing, 1992.

[8] R.C. Backhouse, P. de Bruin, G. Malcolm, T.S. Voermans, and J. van der Woude. Relational catamorphisms. In Bernhard Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers B.V., 1991.

[9] Roland Backhouse and Jeremy Gibbons. Programming with type classes. Presentation at WG2.1#55, Bolivia, January 2001.

[10] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

[11] Andrew Bromage. Haskell Foundation Library. `www.sourceforge.net/projects/hfl/`, 2002.

[12] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Universiteit Utrecht, 2001.

[13] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[14] Peter Drayton, Ben Albahari, and Ted Neward. *C♯ in a Nutshell*. O'Reilly, 2002.

[15] Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Dept INF, Univ Twente, June 1994.

[16] Richard P. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1996.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] Jeremy Gibbons. Origami programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*. Palgrave, 2003.

[19] Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming*. Kluwer Academic Publishers, 2003.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[21] Douglas Gregor and Sybille Schupp. Making the usage of STL safe. In Gibbons and Jeuring [19].

[22] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43:129–159, 2002. Earlier version appears in LNCS 1837: Mathematics of Program Construction, 2000.

[23] Ralf Hinze and Johan Jeuring. Weaving a web. *Journal of Functional Programming*, 11(6):681–689, 2001.

[24] Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.

[25] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[26] John Hughes. Why functional programming matters. *Computer Journal*, 1989.

[27] Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–72, 2002.

[28] Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, editors. *Generic Programming*. Springer-Verlag, 2000.

[29] Mark P. Jones. *Qualified Types: Theory and Practice*. DPhil thesis, University of Oxford, 1992.

[30] Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4&5):375–392, 2002.

[31] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.

[32] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. In *Symposium on Principles of Programming Languages*, pages 233–244, 2002.

[33] Chris Okasaki. An overview of Edison. Haskell Workshop, 2000.

[34] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[35] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop*, 2002.

[36] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.

[37] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.

[38] Erwin Unruh. Prime number computation. ANSI X3J16-94-0075/ISO WG21-462, 1994.

[39] Todd Veldhuizen. Five compilation models for C++ templates. In *First Workshop on C++ Template Programming*, October 2000.

Already published:

Modern Methods and Algorithms of Quantum Chemistry - Proceedings
Johannes Grotendorst (Editor)
Winter School, 21 - 25 February 2000, Forschungszentrum Jülich
NIC Series Volume 1
ISBN 3-00-005618-1, February 2000, 562 pages
out of print

Modern Methods and Algorithms of Quantum Chemistry - Poster Presentations
Johannes Grotendorst (Editor)
Winter School, 21 - 25 February 2000, Forschungszentrum Jülich
NIC Series Volume 2
ISBN 3-00-005746-3, February 2000, 77 pages
out of print

Modern Methods and Algorithms of Quantum Chemistry - Proceedings, Second Edition
Johannes Grotendorst (Editor)
Winter School, 21 - 25 February 2000, Forschungszentrum Jülich
NIC Series Volume 3
ISBN 3-00-005834-6, December 2000, 638 pages

Nichtlineare Analyse raum-zeitlicher Aspekte der hirnelektrischen Aktivität von Epilepsiepatienten
Jochen Arnold
NIC Series Volume 4
ISBN 3-00-006221-1, September 2000, 120 pages

**Elektron-Elektron-Wechselwirkung in Halbleitern:
Von hochkorrelierten kohärenten Anfangszuständen
zu inkohärentem Transport**
Reinhold Lövenich
NIC Series Volume 5
ISBN 3-00-006329-3, August 2000, 146 pages

Erkennung von Nichtlinearitäten und wechselseitigen Abhängigkeiten in Zeitreihen
Andreas Schmitz
NIC Series Volume 6
ISBN 3-00-007871-1, May 2001, 142 pages

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Kei Davis, Yannis Smaragdakis, Jörg Striegnitz (Editors)
Workshop MPOOL, 18 May 2001, Budapest
NIC Series Volume 7
ISBN 3-00-007968-8, June 2001, 160 pages

Europhysics Conference on Computational Physics - Book of Abstracts
Friedel Hossfeld, Kurt Binder (Editors)
Conference, 5 - 8 September 2001, Aachen
NIC Series Volume 8
ISBN 3-00-008236-0, September 2001, 500 pages

NIC Symposium 2001 - Proceedings
Horst Rollnik, Dietrich Wolf (Editors)
Symposium, 5 - 6 December 2001, Forschungszentrum Jülich
NIC Series Volume 9
ISBN 3-00-009055-X, May 2002, 514 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms - Lecture Notes
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
NIC Series Volume 10
ISBN 3-00-009057-6, February 2002, 548 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms- Poster Presentations
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
NIC Series Volume 11
ISBN 3-00-009058-4, February 2002, 194 pages

Strongly Disordered Quantum Spin Systems in Low Dimensions: Numerical Study of Spin Chains, Spin Ladders and Two-Dimensional Systems
Yu-cheng Lin
NIC Series Volume 12
ISBN 3-00-009056-8, May 2002, 146 pages

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Jörg Striegnitz, Kei Davis, Yannis Smaragdakis (Editors)
Workshop MPOOL 2002, 11 June 2002, Malaga
NIC Series Volume 13

ISBN 3-00-009099-1, June 2002, 132 pages

**Quantum Simulations of Complex Many-Body Systems:
From Theory to Algorithms - Audio-Visual Lecture Notes**
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre,
Kerkrade, The Netherlands
NIC Series Volume 14
ISBN 3-00-010000-8, November 2002, DVD

Numerical Methods for Limit and Shakedown Analysis
Manfred Staat, Michael Heitzer (Eds.)
NIC Series Volume 15
ISBN 3-00-010001-6, February 2003, 306 pages

**Design and Evaluation of a Bandwidth Broker that Provides
Network Quality of Service for Grid Applications**
Volker Sander
NIC Series Volume 16
ISBN 3-00-010002-4, February 2003, 208 pages

**Automatic Performance Analysis on Parallel Computers with
SMP Nodes**
Felix Wolf
NIC Series Volume 17
ISBN 3-00-010003-2, February 2003, 168 pages

**Haptisches Rendern zum Einpassen von hochaufgelösten
Molekülstrukturdaten in niedrigaufgelöste
Elektronenmikroskopie-Dichteverteilungen**
Stefan Birmanns
NIC Series Volume 18
ISBN 3-00-010004-0, September 2003, 178 pages

Auswirkungen der Virtualisierung auf den IT-Betrieb
Wolfgang Gürich (Editor)
GI Conference, 4 - 5 November 2003, Forschungszentrum Jülich
NIC Series Volume 19
ISBN 3-00-009100-9, October 2003, 126 pages

NIC Symposium 2004
Dietrich Wolf, Gernot Münster, Manfred Kremer (Editors)
Symposium, 17 - 18 February 2004, Forschungszentrum Jülich
NIC Series Volume 20
ISBN 3-00-012372-5, February 2004, 482 pages

Measuring Synchronization in Model Systems and Electroencephalographic Time Series from Epilepsy Patients
Thomas Kreutz
NIC Series Volume 21
ISBN 3-00-012373-3, February 2004, 138 pages

Computational Soft Matter: From Synthetic Polymers to Proteins - Poster Abstracts
Norbert Attig, Kurt Binder, Helmut Grubmüller, Kurt Kremer (Editors)
Winter School, 29 February - 6 March 2004, Gustav-Stresemann-Institut Bonn
NIC Series Volume 22
ISBN 3-00-012374-1, February 2004, 120 pages

Computational Soft Matter: From Synthetic Polymers to Proteins - Lecture Notes
Norbert Attig, Kurt Binder, Helmut Grubmüller, Kurt Kremer (Editors)
Winter School, 29 February - 6 March 2004, Gustav-Stresemann-Institut Bonn
NIC Series Volume 23
ISBN 3-00-012641-4, February 2004, 440 pages

Synchronization and Interdependence Measures and their Applications to the Electroencephalogram of Epilepsy Patients and Clustering of Data
Alexander Kraskov
NIC Series Volume 24
ISBN 3-00-013619-3, May 2004, 106 pages

High Performance Computing in Chemistry
Johannes Grotendorst (Editor)
Report of the Joint Research Project:
High Performance Computing in Chemistry - HPC-Chem
NIC Series Volume 25
ISBN 3-00-013618-5, December 2004, 160 pages

All volumes are available online at http://www.fz-juelich.de/nic-series/.