Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 253 (2010) 37–50
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Interfacing Concepts

Why Declaration Style Shouldn't Matter

Anya Helene Bagge[1] Magne Haveraaen[2]

Department of Informatics, University of Bergen, Norway

Abstract

A *concept* (or *signature*) describes the interface of a set of abstract types by listing the operations that should be supported for those types. When implementing a generic operation, such as sorting, we may then specify requirements such as "elements must be comparable" by requiring that the element type models the Comparable concept. We may also use axioms to describe behaviour that should be common to all models of a concept.
However, the operations specified by the concept are not always the ones that are best suited for the implementation. For example, numbers and matrices may both be addable, but adding two numbers is conveniently done by using a return value, whereas adding a sparse and a dense matrix is probably best achieved by modifying the dense matrix. In both cases, though, we may want to pretend we're using a simple function with a return value, as this most closely matches the notation we know from mathematics.
This paper presents two simple concepts to break the notational tie between implementation and use of an operation: *functionalisation*, which derives a set of canonical pure functions from a procedure; and *mutification*, which translates calls using the functionalised declarations into calls to the implemented procedure.

Keywords: concepts, functions, procedures, mutification, axioms, imperative vs functional, concept-based programming

1 Introduction

Concepts is a useful feature for generic programming, allowing programmers to specify the interface and behaviour of abstract data types. The concept feature was introduced to C++0x (the upcoming C++ standard revision) in order to give clearer error messages in templated code, and to provide a way to more easily glue together generic code. Similar features are available in other programming languages.

In this paper we will look at concepts in the context of the Magnolia Programming Language, an experimental language loosely based on C++. Concepts form an integral part of Magnolia, and the programmer is encouraged to specify code

[1] http://www.ii.uib.no/~anya/

[2] http://www.ii.uib.no/~magne/

doi:10.1016/j.entcs.2010.08.030

dependencies in terms of concepts, and use concepts to specify the interface of implementation modules. Concepts are used to hide implementation details, so that one module may be replaced by another modelling the same concept. Even 'standard' data types like numbers and strings are handled through concepts, allowing for the possibility of replacing or using different implementations of basic data types.

If we are to hide away different implementations behind a common interface, we must consider that different situations call for different programming styles. Operations on primitive data types, for example, are conveniently handled using return values, whereas the corresponding operation on a big data structure may be better handled by updating it.

Also, the needs of a library implementation may differ from the needs of the code that uses it. For example, a mathematical problem may be easily expressed with function and operator calls, whereas a communication protocol could be better expressed as a sequence of procedure calls. As a library user[3], however, you are locked to the style the library writer chooses to support – and your preferred style may be less convenient to implement, or less efficient, and thus not supported.

C++ is an example of a language where the proliferation of notational variants is especially bad (or good, depending on your point of view) – with both member and non-member functions, operator overloading, and a multitude of parameter passing modes (value, reference, `const` reference, pointers). Functional languages like ML and Haskell are less problematic since the languages are already restricted to functional-style declaration forms.

Generic and generative techniques are hampered by a proliferation of declaration forms (prototypes, in C++ terminology) for implementations of the same abstract idea. For instance, implementing generic algorithms, such as a generic `map` operation for arrays, is made more difficult when the declaration forms of the element operations vary – so we end up with a multitude of different `map` implementations, e.g., one for arrays of numbers and one arrays of matrices.

For C++0x, the problem of having many different declaration forms for what is essentially the same operation is solved by allowing the implementer to add explicit glue code in the *concept map* (a declaration that some given types model a given concept). In constrained template code, the user will use the declarations from the concept and need not know anything about how the implementation is declared. Note, however that this convenience is available in constrained template code only.

In this paper, we show how this is solved in Magnolia by separating the *implementation signature* from the *use signature*, allowing different call styles to be used independently of the implementation style. By letting the compiler translate between different call styles, we gain increased flexibility for program processors – of both the human and software variety. We call these translations *functionalisation* and *mutification*.

We will start out by giving an intuition about our method, before we define

[3] *User* refers to the programmer who is using a library, and the word *implementer* to the programmer who has implemented it. The *end user* who runs the finished software product is outside the scope of our discussion.

it more formally later in the paper. We illustrate our points with examples in Magnolia, which has functionalisation and mutification as basic, built-in features.

Magnolia supports imperative *procedures*, which are allowed to update their arguments, and pure *functions*, which are not. A sample Magnolia procedure declaration looks like this:

```
procedure fib(upd int n);
```

The procedure `fib` takes a single parameter, an ***upd****atable integer*, which is used both for input and for returning a result.

Functionalisation takes a procedure declaration, and turns it into a function declaration. The functionalisation of `fib` is:

```
function int fib(int n);
```

– a function with an integer parameter, returning an integer. This is the declaration that would typically be used in a concept. A different declaration of the `fib` procedure – like this, for example,

```
procedure fib(obs int n, out int r);
```

– which ***obs****erves* the parameter `n` and ***out****puts* the parameter `r`, yields the same functionalisation.

Functionalising the procedure declaration gives us a function we can call from Magnolia expressions, but no implementation of that function. This is where *mutification* comes into the picture. Mutification takes an expression using a functionalised procedure, and turns it into a statement that calls the procedure directly. Here we show a call for each of the declarations above, the functional form (in the middle) can be transformed to either of the procedural calls shown.

```
y = x;
call fib(y);       ←   y = fib(x);         →   call fib(x,y);
```

Why would we want to implement an algorithm with one declaration form and use it with another? As explained above, the implementation side is often dictated by what fits with the problem or algorithm – in-place sorting and updating matrix operations, for instance, will require less memory and may run faster than versions that construct new objects. There are several reasons why an algebraic style is useful on the user side:

Flexibility: Multiple implementations with different characteristics can be hidden behind the same interface. This is particularly useful in generic code and when building a basis of interchangeable components.

Ease of reasoning: Not just for humans, but also for compilers and tools. For example, axiom-based rewrite rules [2,8,12] allow programmers to aid the compiler's optimisation by embedding simplification rules within a program. Such rules are much harder to express with statements and updating procedures.

Notational clarity: certain problems are most clearly expressed in an algebraic or functional style, which is close to the notational style of mathematics, a notation developed over the centuries for clarity of expressions. Many formal specification languages, e.g., the Larch Shared Language [6] and CASL [9] use the functional style due to its clarity. A program written in an algebraic style is easy to relate to a formal specification.

Using a single declaration form makes it possible to state rules such as "+ and × are commutative for algebraic commutative rings", and have it apply to all types satisfying the properties of commutative rings (i.e., all types modelling the concept *CommutativeRing*) – independently of whether the implementation uses return values or argument updates.

We may in principle choose any declaration form as the common, canonical style – we have chosen the algebraic style for the reasons above, and because it is less influenced by implementation considerations (such as the choice of which argument(s) should output the result). As we shall see, this choice does not mean that we must enforce an algebraic programming style.

We will now proceed with a deeper discussion of the matter. The rest of this paper is organised as follows: First, we introduce the necessary features of the Magnolia language (Sect. 2). Then we define mutification and functionalisation, and explain how they are applied to Magnolia programs (Sect. 3). We continue by discussing some limitations and pragmatic considerations, and the benefits and possibilities of the approach (Sect. 4).

2 The Magnolia Language

This section gives a brief overview of the Magnolia Programming Language. Magnolia is based on C++, with some features removed, some features added, and some changes to the syntax. We have designed it to be easier to process and write tools for than C++ (this is actually our main reason for using a new language, rather than working with C++), while being similar enough that we can easily compile to C++ code and use high-performance C++ compilers.

The following features are the ones that are relevant for this paper:

Procedures have explicit control over their inputs and outputs: they are allowed to modify their parameters, according to the *parameter modes* given in the procedure declaration. The available modes are `obs`*erve* for input-only arguments, `upd`*ate* for arguments that can be both read and written to, and `out`*put* for output-only arguments. These modes describe the data-flow characteristics of the procedure – which values the result may depend on, and which variables may be changed by the procedure. The parameter passing mode (e.g., by value, by reference, by copying in/out) is left undefined, so the compiler is free to use the most efficient passing mode for a given data type. Procedures have no return values, the result is given by writing to one or more arguments. Procedure calls use the `call` keyword, so they are easy to distinguish from function calls.

Functions have a single return value, which depends solely on the arguments. They are not allowed to modify arguments. Operators can be overloaded, and are just fancy syntax for function calls.

Functions and procedures are known collectively as *operations*.

Data types are similar to C++ structs and classes, though there are no member operations – everything is treated as non-members. There is no dynamic dispatch or inheritance yet, and we won't consider that in this paper.

Concepts describe the interfaces of types by listing some required operations, axioms on the operations and possibly other requirements. A set of types is said to model a concept if the operations are defined for those types and the requirements are satisfied. For example, the following defines a simple 'Indexable' concept:[4]

```
concept Indexable(A,I,E) {
  E getElt(A, I);
  A setElt(A, I, E);

  axiom getset(A a, I i, E e) {
    assert getElt(setElt(a, i, e), i) == e;
  }
```

Indexable has three types, and array-like type `A`, an index type `I` and an element type `E`. The concept defines two functions, `getElt` and `setElt`, and a simple axiom relating the functions.

Generics: Generic programming is done through a template facility similar to C++'s. Type parameters may be constrained using concepts, so that only types modelling a given concept are acceptable as arguments.

No Aliasing: Aliasing makes it difficult to reason about code, because it destroys the basic assumption that assigning to or updating one variable will not change the value of another. Functional languages avoid this problem by simply banning the whole idea of modifying variables. We feel disallowing modification is too high a price to pay, particularly when working with numerical software and large data structures, so Magnolia has a set of rules designed to prevent aliasing while still having most of the freedom of imperative-style code:

- No pointers or references. Any data structures that need such features must be hidden behind a 'clean' interface. The programmer must take responsibility for ensuring that the implementation is safe.
- There is no way to refer to a part of a data structure. For example, you can't pass an element of an array as a procedure parameter – you must either pass the entire array, or the *value* of the element. Thus, changing an object field or an array element is an operation on the object or array itself (unlike in C++, where fields and elements are typically l-values and can be assigned to directly). This is why the `setElt` operation in the Indexable concept above is declared as a function returning an array, and not returning a reference to an element as is typical in C++.
- If a variable is passed as an `upd` or `out` argument to a procedure, that variable cannot be used in any other argument position in the same call.

[4] We have avoided using operators in the example to keep the syntax simple

3 Relating Functions and Procedures

We will now establish a relationship between Magnolia functions and procedures, so that each procedure declaration has a set of corresponding function declarations given by functionalisation (Def. 3.1), and every expression has a corresponding sequence of procedure calls given by mutification (Def. 3.2).

3.1 Functionalisation of Declarations

Definition 3.1 *Functionalisation*, F, maps a procedure declaration to one or more function declarations. This makes procedures accessible from expressions, at the signature level. Since a procedure can have multiple output parameters, and a function can only have one return value, we get one function for each output parameter of the procedure (numbered 1 to i):

$$\mathrm{F_i}(\mathsf{Proc}(n, \boldsymbol{q})) = \mathsf{Fun}(n_i, \mathrm{Out}(\boldsymbol{q})_i, \mathrm{In}(\boldsymbol{q})) \tag{1}$$

For clarity, we use abstract syntax in the definitions, with Proc (name, proc-parameter-list) being a procedure declaration, and Fun (name, return-type, fun-parameter-list) being a function declaration. In and Out gives the input and output parameters of a procedure, respectively[5]:

$$\mathrm{In}(\boldsymbol{q}) = [t \mid \langle m, t\rangle \leftarrow \boldsymbol{q}, m \in \{\texttt{obs}, \texttt{upd}\}]$$
$$\mathrm{Out}(\boldsymbol{q}) = [t \mid \langle m, t\rangle \leftarrow \boldsymbol{q}, m \in \{\texttt{out}, \texttt{upd}\}]$$

where m is the parameter mode and t is the parameter type.

We can then obtain the list of functions corresponding to a procedure:

$$\mathrm{F}(\mathsf{Proc}(n, \boldsymbol{q})) = [\mathrm{F}_i(\mathsf{Proc}(n, \boldsymbol{q})) \mid i = 1 \ldots \mathrm{len}(\mathrm{Out}(\boldsymbol{q}))] \tag{2}$$

Note the similarity between functionalisation and standard techniques for describing the semantics of a procedure with multiple return values. This is the link between the semantics of the procedure and the functionalised version, and the key in maintaining semantic correctness between the two programming notations.

For example, the following procedures:

```
procedure plus(upd dense x, obs sparse y);
procedure plus(obs int x, obs int y, out int z);
procedure copy(obs T x, out T y);
```

functionalise to the following functions:

```
function dense plus(dense x, sparse y);
function int plus(int x, int y);
function T copy(T x);
```

[5] Using list comprehension notation (similar to set notation), as in Haskell or Python.

which is what would be used in a concept declaration. We keep the namespaces of functions and procedures, as well as their usage notations (expressions versus calls), distinct, thus avoiding overloading conflicts. For multi-output procedures, the functions get numbered names – Magnolia also allows the programmer to choose the function names, if desired.

Note that the inverse operation – obtaining a procedure from a function – is not straight-forward, since there are many different mode combinations for the same function declaration. However, we could define a *canonical proceduralisation*, P, in which every function is mapped to a procedure with one `out` parameter for the return value and one `obs` parameter for each parameter of the function:

$$\mathrm{P}(\mathsf{Fun}(n, t', [t_1, \ldots, t_n]) = \mathsf{Proc}(n, [\langle \mathtt{obs}, t_1\rangle, \ldots, \langle \mathtt{obs}, t_n\rangle, \langle \mathtt{out}, t'\rangle]) \tag{3}$$

3.2 Mutification

Definition 3.2 [Mutification of Assignment] Mutification turns a sequence of function calls and assignments into a procedure call. Given a procedure $p = \mathsf{Proc}(n, \boldsymbol{q})$:

$$\mathcal{M}(\overline{\mathsf{Assign}(\boldsymbol{y}, \mathsf{Apply}(\boldsymbol{f}, \boldsymbol{x}))}) = \boldsymbol{i} \;;\; \mathsf{Call}(\mathsf{Proc}(n, \boldsymbol{q}), \boldsymbol{x'}),\ \text{ where } \mathrm{F}(\mathsf{Proc}(n, \boldsymbol{q})) = \boldsymbol{f},$$

$$\text{and } \langle \boldsymbol{x'}, \boldsymbol{i}\rangle = \text{unzip}\left(\boldsymbol{q}\begin{bmatrix} \langle \mathtt{obs}, s\rangle \to \langle x, \mathsf{Nop}\rangle \text{ if } x \notin \boldsymbol{y} \\ \qquad\qquad \text{where } x = \downarrow \boldsymbol{x} \\ \langle \mathtt{obs}, s\rangle \to \langle t, \mathsf{TmpVar}(t, x)\rangle \text{ if } x \in \boldsymbol{y} \\ \qquad\qquad \text{where } x = \downarrow \boldsymbol{x} \\ \langle \mathtt{out}, s\rangle \to \langle \downarrow \boldsymbol{y}, \mathsf{Nop}\rangle \\ \langle \mathtt{upd}, s\rangle \to \langle y, \mathsf{Assign}(y, \downarrow \boldsymbol{x})\rangle, \\ \qquad\qquad \text{where } y = \downarrow \boldsymbol{y} \end{bmatrix}\right)$$

$$\text{and } y_j \equiv y_k \iff j = k \text{ for all } y_j, y_k \in \boldsymbol{y} \tag{4}$$

The pattern $\overline{\mathsf{Assign}(\boldsymbol{y}, \mathsf{Apply}(\boldsymbol{f}, \boldsymbol{x}))}$ recognises a sequence of assignments, one for each `upd` or `out` argument of p. The list of functions called, $\boldsymbol{f}$, must match the functionalisation of p, in sequence. Dummy assignments can be inserted to accomplish this, if no suitable instructions can be moved from elsewhere. All variables ($\boldsymbol{y}$) assigned to must be distinct.

We then construct a new argument list $\boldsymbol{x'}$ and a list of setup statements $\boldsymbol{i}$ by examining the formal parameter list of p, picking ($\downarrow$) an argument from $\boldsymbol{x}$ in case of `obs`, picking from $\boldsymbol{y}$ in case of `out`, and picking from $\boldsymbol{y}$ and generating an assignment $\mathsf{Assign}(y, x)$ in case of `upd`. Nop denotes an empty instruction, TmpVar creates a temporary variable, and Assign denotes an assignment. To avoid aliasing problems, a temporary is needed to store the value of an `obs` argument which is also used for output.

Generating assignments for `upd` arguments is necessary, since the variables $\boldsymbol{y}$ may have different values than the corresponding variables in the original argument list. This may generate redundant $\mathsf{Assign}(y, y)$ instructions when y is already in an update position – these can trivially be eliminated at a later stage.

For example, if we have a procedure $p(\texttt{out}\, t_1, \texttt{upd}\, t_2, \texttt{obs}\, t_3)$ and $f_1, f_2 = F(p)$:

```
a = f_1(3, 2);                    b = 3;
b = f_2(3, 2);        →           call p(a, b, 2);
```

3.3 Functionalisation of Procedure Calls

Given a canonical mapping of function declarations to procedure declarations (i.e., *proceduralisation*, as sketched in Sect. 3.1), we can functionalise procedure calls – transform them to function calls and assignments.

This means we lose control over some aspects of the program, such as the creation and deletion of intermediate variables. However, since we argue that expression-based programs are much easier to analyse and process by tools, it would be useful to obtain something as close to a pure expression-based program as possible – even if we will eventually mutify it again to obtain imperative code. Thus, the choice of language form to work on becomes one of convenience. Our experience is mostly with high-level optimisations that take advantage of algebraic laws – which is a prefect fit for algebraic-style Magnolia. The same can be seen in modern optimising compilers, which will typically transform low-level code to and from Static Single Assignment (SSA) form depending on which optimisation is performed.

Definition 3.3 [Functionalisation of Procedure Calls] For a procedure $p = \mathsf{Proc}(n, \boldsymbol{q})$:

$$\mathcal{F}(\mathsf{Call}(p, \boldsymbol{x})) = [\mathcal{F}_i(\mathsf{Call}(p, \boldsymbol{x})) | i = 1 ... \mathrm{len}(\mathrm{Out}(\boldsymbol{q}))] \tag{5}$$

$$\mathcal{F}_i(\mathsf{Call}(p, \boldsymbol{x})) = \mathsf{Assign}(\boldsymbol{y}_i, \mathsf{Apply}(\mathrm{F}_i(p), \boldsymbol{x}')) \tag{6}$$

where $\boldsymbol{y} = \mathrm{Out}_{\boldsymbol{q}}(\boldsymbol{x})$ and $\boldsymbol{x}' = \mathrm{In}_{\boldsymbol{q}}(\boldsymbol{x})$. The ordering of assignments is immaterial, since the requirements on procedure call arguments ensures that the variables $\boldsymbol{y}$ are distinct from $\boldsymbol{x}'$.

To functionalise a procedure call $\mathsf{Call}(n, \boldsymbol{q})$, we build a new argument list $\boldsymbol{x}'$ of all the input arguments, then generate one assignment for each output argument. $\mathrm{In}_{\boldsymbol{q}}$ and $\mathrm{Out}_{\boldsymbol{q}}$ select the input and output arguments of an actual argument list with respect to a formal parameter list $\boldsymbol{q}$:

$$\mathrm{In}_{\boldsymbol{q}}(\boldsymbol{x}) = [x \mid \langle x, \langle m, t \rangle\rangle \leftarrow \langle \boldsymbol{x}, \boldsymbol{q}\rangle \text{ if } m \in \{\texttt{obs}, \texttt{upd}\}]$$
$$\mathrm{Out}_{\boldsymbol{q}}(\boldsymbol{x}) = [x \mid \langle x, \langle m, t \rangle\rangle \leftarrow \langle \boldsymbol{x}, \boldsymbol{q}\rangle \text{ if } m \in \{\texttt{out}, \texttt{upd}\}]$$

For example, given a `function int f(int)`, we may use the canonical proceduralisation (3) to obtain and use a `procedure p(obs int, out int)`. Since (3) only gives us single-output procedures, we will only get a single assignment for each call to p. Using (5) above, we can functionalise the following statements:

```
call p(5, x);                     x = f(5);
call p(x, y);         →           y = f(x);
```

Data-flow analysis will allow us to transform the code to `y = f(f(5))`, possibly eliminating the x if it is not needed.

3.4 Mutification of Whole Programs

Mutification has strict assumptions about the instructions it operates on, so we may need to perform instruction reordering and manipulation to make a program suitable for mutification:

Nested expressions must be broken up. This is done by moving a sub-expression out of its containing expression and assigning it to a temporary variable. When functionalising a program, the reverse operation (expression inlining) can be applied to make as deeply nested expressions as possible, thus enabling easy application of high-level transformation rules.

Calls in control-flow statements must be moved outside. Mutification can't be applied directly to calls in conditions, `return`-statements and so on. Introducing a temporary, as above, we replace the call with a reference to the variable (taking care to recompute the value of the variable for each loop iteration in the case of loops).

Multi-valued procedures should be specialised. If we can't fill up all the outputs of a multi-valued procedure, we create a *sliced* version, with only the needed outputs, and with any unnecessary computations removed.

Instructions should be reordered to take advantage of mutification to a multi-valued procedure call. In general, if the instruction i_2 does not depend on the result of i_1, and does not change variables in i_1, it can be moved in front of i_1.

If reordering fails and slicing is also impossible – for example, if the procedure implementation isn't available – we must insert a dummy call with a throw-away result.

Instructions may be made independent of each other by introducing a temporary. For example, if we want to move the second instruction in front of the first, we can store the value of y in a temporary variable:

```
                       int t = y;                int t = y;
x = f(y);          →   x = f(t);           →     y = g(3);
y = g(3);              y = g(3);                 x = f(t);
```

Dig up your old compiler book! Instruction reordering and scheduling is well-known from compiler construction, and similar techniques can be applied here. For example, reordering assignments to take advantage of a multi-valued procedure call is not unlike reordering instructions to take advantage of parallel execution paths in a processor.

Many familiar optimisation techniques like constant propagation, value numbering, common sub-expression elimination, dead variable/code elimination and so on can readily be applied to Magnolia programs, either by first converting to a SSA (static single assignment) form, or by using simple data-flow analysis [10]. Pure functions and explicit information about input and outputs of procedures makes it a lot easier to apply optimisations.

Fig. 1 shows an example program before and after mutification, using the actual intermediate output of the mutification stage of the compiler.

```
procedure fib(out int f, obs int n)
{
  if(n < 2)
    f = n;
  else
    f = fib(n-1) + fib(n-2);
}
```

```
procedure fib (out int f, obs int n) {
  bool a_0;
  call _<_(a_0, n, 2);
  if(a_0)
    f = n;
  else
    int c_0;
    c_0 = n;
    call _-_(c_0, 2);
    int b_0;
    call fibonacci::fib(b_0, c_0);
    int d_0;
    d_0 = n;
    call _-_(d_0, 1);
    call fibonacci::fib(f, d_0);
    call _+_(f, b_0);
}
```

```
procedure fib2(upd int n)
{
  if(n < 2)
    ;
  else
    n = fib2(n-1) + fib2(n-2);
}
```

```
procedure fib2 (upd int n) {
  bool e_0;
  call _<_(e_0, n, 2);
  if(e_0)
    ;
  else
    int f_0;
    f_0 = n;
    call _-_(f_0, 2);
    call fibonacci::fib2(f_0);
    call _-_(n, 1);
    call fibonacci::fib2(n);
    call _+_(n, f_0);
}
```

Fig. 1. Left: Two different variants of a recursive Fibonacci procedure. Right: Results of mutifying the two procedures (actual intermediate output from the compiler). Note that `fib2` – using an `upd` parameter – requires fewer temporaries than `fib`, which uses an `out`/`obs` combination. The notation `_+_(a,b)` is simply a desugared function-call variant of an operator call `a + b`, and `call _+_(a,b)` is an updating procedure call, similar to `a += b` in C++.

4 Discussion

4.1 Working with Concepts and Axioms

The main benefit we have seen so far (apart from the somewhat fuzzily defined notational clarity) is in the relationship between code and specification, particularly with the concept feature.

- Most operations defined in a concept will have an algebraic-style declaration (with the exception of certain things like I/O). There is no need for deciding whether to define a functional-style interface, or an OO-style interface or an imperative-style interface.
- Axioms defined in concepts map directly to algebraic specifications. In C++, axioms for imperative-style concepts are typically written using the comma operator (sequential composition of expressions):

```
axiom getset(A a, I i, E e) {
  setElt(a,i,e), getElt(a,i) == e;
}
```

- In most cases, there is a built-in well-defined mapping between procedure declarations and the function declarations used in a concept. As long as program code is written against the concept interfaces, axioms are easily related to program code. In C++, small wrappers are often required to translate between the declaration style in the concept and that of the implementation. This indirection makes it much harder to relate axioms to actual program code, for example for use as rewrite rules [12].

Axiom-based rewriting [2] is a convenient way to do high-level optimisation by using equational axioms as rewrite rules – similar to how algebraic laws are used to simplify arithmetic expressions. With code based on pure, well-behaved functions and no aliasing, this is very simple to implement, and rules can be applied without any of the complicated alias analysis that are part of modern compilers.

Note that while this work complements the concept feature, it is by no means dependent on it, and will work equally well without.

4.2 Limitations of Mutification

Mutification and functionalisation have some limitations. We are basically hiding imperative code behind an interface of pure functions, and if we are to do this without costly overhead, what we're hiding must be reasonably pure. This means that:

- Procedure results can only be influenced by input arguments.
- A procedure can have no other effect on program state than its effect on its output arguments.
- Objects used as function arguments must be *clonable* – i.e., it must be possible to save and restore the complete state of the object.

The two former limitations rule out procedures operating on global variables. The latter rules out things like stream I/O and user interaction.

Global variables are problematic because they interfere with reasoning – introducing unexpected dependencies or causing unexpected side-effects. This breaks the simple algebraic reasoning model that allows us to move code around and modify it without complicated analysis. Global variables can be handled in some cases by passing them as arguments, either explicitly or implicitly (having the compiler add global variables to declarations and calls). At some point we will reach an outermost procedure in the call tree which will have to either get all globals as arguments from the run-time system, or be allowed to access globals directly. If the global state is large, passing it around can be impractical, particularly since mutification will require that the state can be cloned when necessary. However, global variables are generally frowned upon anyway, and create problems with reentrancy and multi-threading, so this may not be a problem in practice.

The clonability requirement comes from the need to sometimes introduce temporaries while applying mutification, which is necessary to protect against aliasing, and also useful to avoid unnecessary recomputations. Some objects are however unclonable, e.g., because they represent some kind of interaction with the real world (reading and writing to a file or terminal, for example). Such objects are considered *impure*. They can only be used as `upd` arguments, and transformation on code involving impure objects must preserve the order of and conditions under which the objects are accessed. This is similar to how a C compiler would treat `volatile` variables in low-level code like device drivers.

Impure objects are best handled using procedures, though in principle we can mutify function calls involving impure objects as long as no copying is required.

4.3 Performance

We have no performance data on Magnolia yet, but we have experimented with a simple form of mutification to reduce overhead in the Sophus numerical software library [3,7]. Sophus is implemented in C++ in an algebraic coding style – i.e., preferring pure function calls over argument updates. Mutifying the code of the Seismod seismic simulation application, we saw a speedup factor of 1.8 (large data sets) to 2.0 (small data sets) compared to the original non-mutified algebraic-style code. Memory usage was reduced to 60%.

4.4 Related Work

Fluent languages [4] combine functional and imperative styles by separating the language into sub-languages, with *PROCEDUREs* which are fully imperative, *OBSERVERs*, which do not cause side-effects, *FUNCTIONs*, which do not cause and are not affected by side-effects, *PUREs* which are referentially transparent (no side-effects, and return the same value on every evaluation). The invariants are maintained by forbidding calls to subroutines with more relaxed restrictions. Our mutification, on the other hand, takes responsibility for protecting against harmful side-effects, allowing calls to procedures from functions.

The Euclid language [11] is designed with verification in mind, and has the same distinction between procedures and side-effect free functions as Magnolia, and also forbids aliasing. The aliasing rules are similar to Magnolia, but also deal with pointers and passing array components and dereferenced pointers as arguments, which is forbidden in Magnolia.

Mutification bears some resemblance to the translation to three-address code present in compilers [1]. Our approach is more high-level and general, dealing with arbitrary procedures instead of a predetermined set of assembly/intermediate-language instructions.

Copy elimination [5] is a method used in compilation functional languages to avoid unnecessary temporaries. By finding an appropriate target for each expression, evaluation can store the results directly in the place where it is needed, thus eliminating (at least some) intermediate copies.

Expression templates [13] avoid intermediates and provide further optimisation by using template meta-programming in C++ to compile expressions (particularly large-size numerical expressions) directly into an assignment function. This allows the user to write algebraic-style expressions, but an extra burden is placed on the implementer who must manually code all operators into the expression template system.

5 Conclusion

We have presented a programming method and formal tool, which decouples the usage and implementation of operations. Through *functionalisation* and *mutification*, we make imperative procedures callable as algebraic-style functions and provide a translation between code using function calls and code using procedure calls. This decoupling brings us the following benefits:

- Reuse—having a unified call style simplifies the interfacing of generic code with a wide range of implementations. Signature manipulation may also help integrate code from different sources. This is by no means a complete solution to the problem of reuse, but a small piece in the puzzle which may make things easier.
- Flexibility—multiple implementations with different performance characteristics can be accessed through the same interface.
- Notational clarity of functional style – algebraic notation is similar to mathematics and well suited to express mathematical problems.
- Link to axioms and algebraic specification. The algebraic notation can be directly related to the notation used in specifications, thus bridging the gap between practical programming and formal specifications. This enables the use of axioms for high-level optimisations and for automated testing [2].
- The procedural imperative style with *in situ* updating of variables provide better space and time efficiency, particularly for large data structures.

The notational and reasoning benefits are similar to what is offered by functional programming languages, without requiring immutable variables.

Our initial experiments with mutification were done on C++, because the excellent, high-performance compilers available for C++ make it a good choice for performance-critical code. Languages like C++ are however notoriously difficult to analyse and process by tools. The idea behind Magnolia is to cut away the parts of C++ that are difficult to process or that interfere with our ability to reason about the code, and then add the features necessary to support our method. The Magnolia compiler will produce C++ code, allowing us to leverage both good compilers and good programming methodology.

This paper expands on earlier work [3] on mutification by (1) allowing the imperative forms to update an arbitrary number of arguments (previously limited to one), (2) allowing functionalisation of procedure calls, (3) providing a formal treatment of functionalisation and mutification, and (4) doing this independently of the numerical application domain which was the core of [3]. Functionalisation and mutification was originally motivated by code clarity and efficiency concerns, with the benefits to generic programming becoming apparent later on.

More work remains on determining the performance improvement that can be expected when using imperative vs. algebraic styles, and the productivity improvement that can be expected when using algebraic vs. imperative style. Both these aspects are software engineering considerations. Further work on formally proving the effectiveness and correctness of mutification is also needed.

A prototype implementation of Magnolia, supporting functionalisation and mutification, is available at: http://magnolia-lang.org

References

[1] Aho, A. V., R. Sethi and J. D. Ullman, "Compilers — Principles, Techniques, and Tools," Addison-Wesley, 1986.

[2] Bagge, A. H. and M. Haveraaen, *Axiom-based transformations: Optimisation and testing*, in: J. Vinju and A. Johnstone, editors, *Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, ENTCS (2008).

[3] Dinesh, T., M. Haveraaen and J. Heering, *An algebraic programming style for numerical software and its optimization*, Scientific Programming **8** (2000), pp. 247–259.

[4] Gifford, D. K. and J. M. Lucassen, *Integrating functional and imperative programming*, in: *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming* (1986), pp. 28–38.

[5] Gopinath, K. and J. L. Hennessy, *Copy elimination in functional languages*, in: *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989), pp. 303–314.

[6] Guttag, J. and J. Horning, *Report on the Larch shared language*, Science of Computer Programming **6** (1986), pp. 103–134.

[7] Haveraaen, M., H. A. Friis and T. A. Johansen, *Formal software engineering for computational modelling*, Nordic Journal of Computing **6** (1999), pp. 241–270.

[8] Jones, S. P., A. Tolmach and T. Hoare, *Playing by the rules: Rewriting as a practical optimisation technique in GHC*, in: R. Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, 2001.

[9] Mosses, P. D., editor, "CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language," LNCS **2960**, Springer-Verlag, 2004.

[10] Olmos, K. and E. Visser, *Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules*, in: R. Bodik, editor, *14th International Conference on Compiler Construction (CC'05)*, Lecture Notes in Computer Science **3443** (2005), pp. 204–220.

[11] Popek, G. J., J. J. Horning, B. W. Lampson, J. G. Mitchell and R. L. London, *Notes on the design of Euclid*, in: *Proceedings of an ACM conference on Language design for reliable software*, 1977, pp. 11–18.

[12] Tang, X. and J. Järvi, *Concept-based optimization*, in: *Proceedings of the ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07)* (2007).

[13] Veldhuizen, T. L., *Expression templates*, C++ Report **7** (1995), pp. 26–31, reprinted in C++ Gems, ed. Stanley Lippman.