Scrap++: Scrap Your Boilerplate in C++

Gustav Munkby Andreas Priesnitz Sibylle Schupp Marcin Zalewski
Dept. of Computing Science, Chalmers University of Technology, Göteborg, Sweden.
fmunkby,priesnit,schupp,zalewskig@cs.chalmers.se

Abstract

"Scrap Your Boilerplate" (SYB) is a well studied and widely used design pattern for generic traversal in the Haskell language, but almost unknown to generic programmers in C++. This paper shows that SYB can be implemented in C++. It identifies the features and idioms of C++ that correspond to the Haskell constructs that implement SYB, or can be used to emulate them, and provides a prototype C++ implementation.

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Generic Programming; D.2.13 [*Software Engineering*]: Reusable Software; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Generic programming, C++, extensibility

1. Introduction

"Scrap your boilerplate" (SYB) [18] is a technique for generic queries and traversal that was introduced in Haskell and further refined there [10, 11, 16–18]; for many Haskell programmers, the SYB library Data.Generics has become a standard tool. Despite the apparent usefulness of the SYB library in Haskell, no comparable library is available to the C++ community. Since the current incarnation of SYB in Haskell depends on type classes, higher-order functions, pattern matching, and other features that are not available in C++, we wondered how, if at all, the counterpart of SYB in C++ looks. In this paper, we present and discuss our implementation of the original SYB version [18] in generic C++.

Generic programming in C++ depends on templates, which, unlike polymorphic functions in Haskell, cannot be type-checked when defined, but only when instantiated. Given the differences between the two languages, our C++ solution corresponds surprisingly closely to its Haskell counterpart: each ingredient of the SYB approach can be mapped to C++. However, C++ introduces complications not present in Haskell. For one, certain type checks in Haskell require explicit type computations in C++, which have to be provided as template metaprograms. Although parts of these computations can be automated in libraries [3, 5, 19, 25], type checking becomes more inconvenient than in Haskell. Another important aspect in C++ concerns the difference between in-place and out-of-place computations that are sometimes difficult to unify in the same implementation—not an issue in Haskell, a pure, functional language. On the other hand, some parts of SYB are easier to implement in C++; the best example is the type extension of functions discussed in Sect. 3.3. In Haskell, at least one language extension is necessary to support SYB; as we will see, in C++, no extensions are needed. In this paper we stay within the generic programming paradigm as understood in C++, thus do not consider object-oriented design alternatives.

The rest of the paper is organized in a conceptually top-down manner. In Sect. 2 we briefly introduce the idea behind SYB using the standard "paradise" example from the original SYB paper [18]. In Sect. 3 we discuss the design issues and present an outline of the C++ implementation: recursive traversal combinators in Sect. 3.1, one-layer traversal in Sect. 3.2, and type extension in Sect. 3.3. In Sect. 4 we detail what has to be done for each type to make it SYB compatible and in Sect. 5 we present one possible way of automating the task. We reflect on the scope of SYB in the context of class-based languages in Sect. 6 and discuss the remaining features of SYB not present in our current implementation along with possible generalizations in Sect. 7. In Sect. 8 we summarize related work. We conclude in Sect. 9 with a summary and an outlook on future work.

All Haskell code is taken from the first SYB paper [18]. For brevity, C++ code is provided in a simplified form omitting irrelevant technical details, for example, the `inline` specifier. While the Haskell code has been in use and tested for some time, the C++ code is for exposition; our current prototype does not claim to provide an industrial-strength solution.

2. The Paradise Example

SYB problems deal with applying local computations to tree-like structures. In the original SYB paper, the motivating "paradise" example is about increasing the salaries of all employees in a company. The type representing a company reflects a tree-like company structure: each company has subunits, each subunit has employees and managers, each employee is a person that has a salary, and so on; the local computation in that example is the increase of a salary.

The straight-forward implementation of the salary increase involves just a single function, `incS`, containing the type-specific computation for increasing a salary. To apply this function to a company, however, a lot of "boilerplate" code is required just to extract the relevant bits, namely the salaries within the company. In the SYB literature, the type-specific computation `incS` often is informally referred to as "the interesting" computation.

The SYB solution defines *generic traversal combinators* that, combined with `incS`, form a type-generic function `increase`. Applying `increase` to a company will traverse its tree-like structure and apply `incS` to all salaries:

#1 Haskell
```
increase :: Float -> Company -> Company
increase k = everywhere (mkT (incS k))
```

#2 C++
```
Company& increase(float k, Company& company) {
    return everywhere(mkT(incS(k)), company);
}
```

In the body of the increase function, there are two combinators at work. Firstly, the *recursive traversal combinator* everywhere, which traverses a structure and recursively applies a transformation to all parts. Secondly, the *generic transformation combinator* mkT, which makes a type-specific transformation type-generic, by applying the identity transformation to other types.

In the following section, we elaborate on each of the constituents of the SYB approach and compare the Haskell and C++ implementations.

3. SYB in C++

The SYB approach to generic traversal consists of three major ingredients: recursive traversal combinators, a one-layer traversal, and a type extension of the "interesting" functions. In this section we discuss the issues to be considered when implementing SYB in C++. The discussion of each part of SYB is accompanied by snippets of code in Haskell and C++. As said before, the C++ code presents an illustration of the discussion rather than a final solution; the complete listing can be found in the appendix.

3.1 Recursive Traversal

Conceptually speaking, the most characteristic functions in the SYB approach are generic traversal strategies; in the paradise example in Sect. 2, the strategy everywhere was used. Traversal strategies combine one-layer traversal with an "interesting" function to obtain a task-specific recursive traversal. The strategy everywhere, for example, takes an arbitrary transformation and a value, and applies this transformation to all nodes of the tree-like structure of the value. In Haskell, everywhere simply is a combinator that constructs a recursive traversal.

To implement everywhere in C++, we need to decide how to represent the recursive traversal. Recursive traversal requires passing as argument the composition of everywhere and the transformation to be applied. Like combinators in general, it implies the existence of higher-order functions, which have no direct support in C++. The common approximation to higher-order functions in C++ are *function objects*, which are classes that define the so-called function-call operator (a.k.a. parentheses operator). Instances of a function object can be passed to functions much as any other object, but behave like functions when their function-call operator is called. Moreover, function objects permit function composition by taking another function object as constructor argument and then defining the body of their function-call operator appropriately.

Since we can assume that we know at instantiation time with which transformation everywhere is composed, we represent everywhere in C++ as a function-object template, more precisely, a function object with a static type parameter and a corresponding dynamic constructor parameter; we call this function object Everywhere. At instantiation time its static parameter is bound to the type of the particular transformation function object. When an instance of everywhere is created, the actual function object is passed as an argument to the constructor.

Because C++ has no type signatures for template parameters, there is no way to directly specify that the transformation function passed to Everywhere must be polymorphic and to type-check function arguments against a type signature. Instead, a C++ compiler type-checks templates when they are instantiated. In the general case, the transformation passed to Everywhere will be applied to terms of different types. Therefore, if the transformation is not polymorphic and cannot be applied to terms of different types, the type check fails—in effect, Everywhere encodes a rank-2 polymorphic function without having a rank-2 type.

The code for the everywhere combinator in each of the two languages follows. In both examples, much of the work is forwarded to the function gmapT. This function performs a specific kind of one-layer traversal, namely the transformation of a term; gmapT will be defined in Sect. 3.2.2. In Sect. 2 we have shown how to use everywhere in Haskell and in C++, in the appendix we list a complete main program.

#3 Haskell
```
everywhere :: Data a
           => (forall b. Data b => b -> b)
           -> a -> a
everywhere f x = f (gmapT (everywhere f) x)
```

#4 C++
```
template <typename Fun_>
struct Everywhere
{
    Fun_ fun;
    Everywhere(Fun_ fun) : fun(fun) {}

    template <typename Param_>
    Param_& operator()(Param_& param)
    {
        return fun(gmapT(*this,param));
    }
};

template <typename Fun_, typename Param_>
Param_& everywhere(Fun_ fun, Param_& param) {
    return (Everywhere<Fun_>(fun))(param);
}
```

As the definition of the class Everywhere shows, the Haskell combinator everywhere is realized in C++ as a small class template, consisting of a constructor and one member function, the function-call operator. The class stores the transformation function in a member variable. When its function-call operator is called, it first applies itself to all subterms of the current term using the gmapT one-layer traversal function and finally applies the stored transformation to the current term, effectively encoding bottom-up traversal. Instances of Everywhere correspond to the composition (everywhere f) in Haskell. Thus, passing of the *this object to gmapT corresponds to the recursive call.

Following a standard idiom, we accompany the class template Everywhere by a function template everywhere, which creates an anonymous instance of the class, calls its function-call operator with the argument param, and returns the result. The function is provided for mere convenience and exploits the fact that C++ performs type inference on arguments to function templates, thus saves users the declaration of variables of class-template type. Finally, we note that the curried Haskell function of two arguments has become a binary function in C++. There are several ways of emulating currying in C++. Our implementation of everywhere stores the transformation function in a member variable. A similar effect is also achievable by using, for example, a bind construct like that from the Boost libraries [6].

3.2 One-Layer Traversal

The definition of the generic traversal strategy relies on the function gmapT to perform one-layer traversal. Other combinators of the SYB approach similarly rely on the functions gmapQ (Q for queries) and gmapM (M for monadic transformation). In the SYB approach, these mappings are abstracted to a generic fold function, gfoldl, which, conversely, allows defining gmapT and other combinators by applying gfoldl appropriately. We first discuss the definition of gfoldl and our corresponding implementation in C++, then one of its possible applications, gmapT.

3.2.1 The Generic Fold

The generic fold in SYB folds a term of some type into a term of a possibly different type: it starts by folding the empty constructor application and then, step by step, combines the folded application with the remaining subterms. For example, gmapT, aside from technical details, preserves the original constructor and applies it to the transformed subterms producing a new term of the same type as the original one. Unlike the usual fold, the subterms that gfoldl folds may be of different types; the first subterm is a special case of an empty constructor application. Since gfoldl has to be general enough to allow folding of a term to a new term of a possibly different type, its type is quite hard to digest even for Haskell professionals. Intuitively speaking, gfoldl takes two polymorphic functions and a term, where its first argument (polymorphically) controls the folding of an empty construction application and the second argument (polymorphically) controls the subsequent combination of the folded “tail” with the next subterm. Because both functions must be polymorphic, the type of gfoldl is rank-2, thus relies on an extension to Haskell 98, which, however, is now well-established. Its full type is given in Ex. # 5 below.

Conceptually, the implementation of gfoldl in C++ is very similar to the one in Haskell, yet, we will refer to it as gfold from now on. The difference in name signifies that in C++ the order of subterms is generally undefined while in Haskell the order is imposed by the constructor application on which gfoldl (left fold) operates. The first argument of the C++ gfold corresponds closely to its Haskell counterpart. For the second argument one has to decide how to pass the information that is contained in a Haskell type-constructor argument. We decided to represent this second argument of gfold as a polymorphic function that takes the type of the current term and a reference to its value as static and dynamic arguments. This function allows one to obtain a type-specific initial value.

Before listing the gfold definition, we point out an important difference between C++ and Haskell. In Haskell, the dependency between the argument types and the return type of a function is captured in the function’s type signature, while in C++ that dependency has to be provided by an explicit type computation—there are no type signatures for template arguments. Thus, to achieve the same effect of a fully polymorphic function one needs to accompany the run-time computation that the function performs by the corresponding compile-time computation for the return type. This was not necessary in the definition of the Everywhere combinator because the return type was constant, but is required in case of gfold where the return type depends not only on the type of the term, but also on the types of all its subterms and the results of applying the supplied functions.

We now list the type of the Haskell gfoldl, then (in pseudocode) the C++ counterpart of the gfoldl function, the gfold function template. The Haskell listing includes the type signature and a specific definition of gfoldl for an Employee type [18], while the C++ listing shows only a specific case since, as pointed out, a type signature cannot be provided.

# 5 Haskell

```
class Typeable a => Data a where
gfoldl ::(forall a b . Data a => w (a -> b)
                             -> a -> w b)
       -> (forall g. g -> w g)
       -> a -> w a
data Employee = E Person Salary
instance Data Employee where
  gfoldl k z (E p s) = (z E ‘k‘ p) ‘k‘ s
```

6 C++

```
template <typename Fun_, typename InitFun_>
typename Result<...>::Type
gfold(Fun_ k, InitFun_ z, Employee& e) {
    return k(k(z(e), e.p), e.s);
}
```

In the definition of gfold, the parameter of type InitFun_ is a function object that creates a type-specific initial value for a particular term; the other two parameters correspond to their Haskell counterparts. The return-type computation mirrors the body of gfold but operates on the corresponding types instead of the values (the technical details are not crucial, but see the appendix for an illustrating definition). The return type of gfold for Employee depends on the type transformations performed by k and z, on the Employee type itself, and on the types of e.p and e.s.

To use gfold properly, it is important to understand that for every function passed as either Fun_ or InitFun_ instance, a type-level computation has to be provided, corresponding to its run-time computation. When using gfold directly, these necessary type computations could require more than average C++ expertise. Typically, however, users employ specific instances of gfold, including the specific maps discussed in Sect. 3.2.2. In these, common cases, the user of an SYB library will not even have to know that such computations occur.

Just as gfoldl, the gfold template has to be defined for every type that supports SYB. In the C++ listing, ad-hoc overloading on the third parameter correlates to the Haskell instance declaration; the definition for the Employee type is discussed in detail in Sect. 4. In addition to function overloading for specific types, the metaprogramming facilities of C++ allow one to specialize gfold for families of types specified by some static predicate; standard techniques include partial specialization or the SFINAE (“substitution failure is not an error”) [14] idiom.

3.2.2 Specific Maps

The generic fold function suffices to express all generic traversals but it is quite complex and not easy to work with. Most commonly performed tasks can be provided in simpler, specialized map functions that are defined in terms of the general fold function. For illustration, we discuss the specialization of the function gmapT, used in the definition of the everywhere combinator in Sect. 3.1. Essentially, gmapT describes a one-layer traversal for transformations, that is, type-preserving mappings.

In Haskell, specializing the generic fold to gmapT requires binding the three parameters of gfoldl appropriately: its second parameter must be bound in a way so that its application preserves the original type (as required for a transformation). Its first parameter, much trickier, must be bound to a binary function that applies a transformation f to its second argument and applies its first argument to the result. The third and unproblematic parameter, finally, is the tree itself. For the Haskell implementation, it follows that the first parameter is bound to the folding function just described and the second parameter to the identity function; a dummy type constructor ID is necessary to avoid the need for a type-level identity function, a feature not supported by Haskell.

Our C++ implementation differs from the Haskell implementation in a crucial aspect. In C++, transformation in-place is a more natural paradigm. To perform an in-place transformation, the folding function passed to gfold as the first argument has to simply apply the transformation to each subterm. By expressing an in-place transformation, gmapT in C++ becomes simpler than in Haskell: our gmapT definition creates a function object that applies the transformation function f to the second argument that represents the current subterm—as in Haskell—but ignores the first argument that represents the folded "tail" of constructor application in Haskell, and just returns that argument. Conceptually, the return type has no meaning in the case of an in-place transformation, but it is necessary because of the gfold definition, which applies its folding function to the already folded part of the current term and the next subterm. Similarly, the initialization function does nothing but to return the current term. The Haskell implementation and a corresponding C++ in-place implementation follow:

# 7 Haskell

```
newtype ID x = ID x
unID :: ID a -> a
unID (ID x) = x
gmapT f x = unID (gfoldl k ID x)
  where
    k (ID c) x = ID (c (f x))
```

8 C++

```
template <typename Fun_, typename Param_>
Param_& gmapT(Fun_ fun, Param_& param) {
    return gfold(ApplyToSecond<Fun_,Param_&>(fun),
                 identity<Param_&>(),
                 param);
}
```

The C++ definition takes and returns a reference to gmapT's parameter so that the transformation is performed in-place and the transformed term is returned. The ApplyToSecond function object is a simple wrapper that takes a unary function object and adapts it to a binary function object. The function call operator applies the unary function object to the second parameter and returns the first parameter, ignoring its value. The Bind function object holds a reference to the object to which gmapT is applied. When called, its argument is ignored and that reference is returned; for the details of the implementation, we refer to the appendix. While the in-place definition does not rely on returning a value, an out-of-place computation requires a return value to abstract gfold from the actual choice at this variation point.

Support for out-of-place transformation is possible as well but is certainly more difficult to accomplish than in-place transformation: while only one way exists to perform in-place transformations, there are different ways to realize out-of-place transformations. Since C++ does not guarantee immutable values, however, any efficient implementation needs to avoid copying of potentially large subtrees. Yet, it is conceivable that one implementation of gmapT can cover both in-place and out-of-place transformation by entirely encapsulating their differences in the appropriately specialized Fun_ parameter. For simplicity, we refrain in this paper from such generalization.

3.3 Type Extension

We have discussed most of the machinery necessary to scrap the boilerplate. The last, crucial step that remains, is to generalize the "interesting" functions so that they can be applied throughout a tree and perform an action on the interesting bits while ignoring the others; in Sect. 2, the mkT function takes such "interesting" function and turns it into a generic transformer. To generalize a non-polymorphic function, a test is necessary to decide if the function can or cannot be applied to the current term.

In the original Haskell SYB approach, type representations have to be compared at run time. If the two types are the same, mkT returns the "interesting" function and a polymorphic identity function otherwise. Because the comparison happens at run time, an explicit run-time type representation is necessary, complemented by an unsafeCoerce operation to perform nominal type cast, that is, a cast without change of representation. A later implementation of type-specific behavior for SYB is performed statically, but relies on introducing appropriate type classes [17].

In C++, where function overloading is statically resolved, the comparable effect of the type-specific cases can be achieved at compile time. By introducing two overloaded functions, which encapsulate the two cases of the type cast—the "interesting" function for one special type and the identity function for all others—and dispatching on the type of the "interesting" function, overload resolution performs the type test automatically and determines at compile time which of the two overloaded functions, thus which of the two type cases, to execute. The implementation of mkT in Haskell and C++ follows:

# 9 Haskell

```
mkT :: (Typeable a, Typeable b)
    => (b -> b) -> a -> a
mkT f = case cast f of
          Just g -> g
          Nothing -> id
```

# 10 C++

```
template <typename Fun_>
class MkT
{
    Fun_ fun_;
public:
    MkT(Fun_ fun) : fun_(fun) {}

    template <typename Param_>
    Param_& operator()(Param_& param)
    {
        return param;
    }
    typename Param1<Fun_>::Type
    operator()(typename Param1<Fun_>::Type param)
    {
        return fun_(param);
    }
};

template <typename Fun_>
MkT<Fun_> mkT(Fun_ fun) {
    return MkT<Fun_>(fun);
}
```

The C++ implementation is a parameterized function object, similarly to the implementation of the Everywhere combinator in Sect. 3.1. The class MkT is a function object initialized with the "interesting" function. The first overloaded function-call operator implements a generic identity, the second operator implements the type-specific application. No overload resolution conflicts can occur since the non-generic operator, if applicable, is always chosen over the generic, template operator by the rules of C++ overload resolution. The type of the first argument of the type-specific operator is extracted from the "interesting" function using a traits template. Trait templates [20], another common idiom, can be compared to a compile-time database that stores type information. The free `mkT` function, finally, is provided as a convenience function for the same reason the `everywhere` function in Sect. 3 accompanied the function object `Everywhere`.

In C++, function overloading is a standard technique to provide generic functions that are specialized for certain types. While a library-provided `mkT` function is useful, it would not be unusual in C++ for the user to specify the type cases by hand, for example, by providing a function that is generalized for families of types rather than specific ones.

4. Defining One-Layer Traversal

Almost all combinators of the SYB pattern are defined type-independently and can therefore be provided by a library. The only combinator that cannot be defined once and for all, is the `gfold` function. As discussed in Sect. 3, `gfold` performs a type-specific traversal. Thus, its definition depends on the particular type of the argument. In the same section, we have outlined how `gfold` can be defined in C++. We now turn to the details of its definition.

Since the behavior of `gfold` is type-specific, it would be natural to provide it as a member function. Yet, we represent `gfold` as a non-member function, which can be overloaded to accept arguments of non-class type, like primitive and pointer types. Defining `gfold` as a free function requires access to possibly private member data of a class, which is granted only if the free function is declared as a `friend` of the class. Thus, `gfold` is defined as a non-member friend function of the class type traversed. As an alternative to friend functions, Ex. # 6 could have been overloaded by a purely global function for arguments of class type, which then invoked a corresponding member function that has data member access. Compared to our solution that is based on friends, however, code complexity would increase. Ex. # 6 illustrated the definition of `gfold` for the `Employee` class. Its corresponding friend declaration is provided below.

# 11 C++

```
class Employee
{
    Person p;
    Salary s;
public:
    friend
    template <typename Fun_, typename InitFun_>
    typename Result<...>::Type
    gfold(Fun_ k, InitFun_ z, Employee& e);
};
```

Friend function declarations must be provided at the time of class definition. Defining the `gfold` function as above, thus, will not work for legacy code—even where the source code is available, so that one can look up all data members of a class, adding a friend declaration to the public interface of a class breaks encapsulation. Unless one is willing to modify legacy code, the recursive tree traversal, thus, ends when an instance of a legacy type is encountered, as if the legacy type was a primitive type. We will take up the questions of legacy code and data member access again in Sect. 6.

Legacy code can be dealt with differently in `Haskell`, as `deriving` clauses may be provided separately from existing type definitions. Unless relying on compiler extensions, though, a developer still has to provide these clauses manually. One might say that one extra function per type is not a prohibitive burden, yet it is a bothersome duty we would like to avoid. Moreover, in C++ one `gfold` definition does not suffice; function parameters can be optionally qualified as `const`, `volatile`, or `const volatile`, and these qualifiers are part of the type check. Any realistic implementation of `gfold` would therefore have to support not just one, but all combinations of the qualifiers for each of its parameters, to preserve all qualification that occur in a generic context. The `gfold` function with its 3 parameters requires 2^3 D 8 different parameter lists, thus 8 different function declarations—ironically with identical body. One might therefore wonder whether it is possible to avoid the need for writing this kind of boilerplate code. The next section will show how to generate `gfold` automatically.

5. Generating Generic Folds

The one-layer maps, as we have seen, must be supplied at least once for every data type. Although the implementation of such functions is relatively straightforward, writing this boilerplate code is still tedious, and automatically generating `gfold` is clearly desirable. A prerequisite for the automatic generation of the `gfold` functions is that the structure of data types is available. One possible approach is to let the compiler generate the `gfold` implementations since it already has the information about the structure of the data types. In Haskell, the SYB pattern was considered important enough to extend the compiler to automatically generate `gfold` for any type. In C++, it is not realistic to assume that all compilers would support a particular extension. However, one can exploit the metaprogramming facilities of the language to generate `gfold` within an ordinary user program.

Conceptually, our approach relies on representing a user-defined type by a *heterogeneous container*, that is, a collection of elements of (possibly) different types. The idea behind this approach is to treat every user-defined type as a compile-time constructed tuple so that this type can be generated in an inductive manner through metaprogramming. In our C++ solution, we represent this inductive construction by the two classes `Insert` and `Empty`; in list-constructor terminology, `Insert` corresponds to `Cons`, `Empty` to `Nil`. Using `Insert` and `Empty`, the `Employee` type can be defined by successively inserting its two members of type `Person` and `Salary`, as demonstrated in the following snippet; conversely, such construction discipline and the use of `Insert` is required for any type for which `gfold` should be automatically generated. Ex. # 12 illustrates the idea; a complete example is listed in the appendix:

# 12 C++

```
typedef Insert<Person,
               Insert<Salary,Empty> > Employee;
```

Because the heterogeneous container is defined in an inductive manner, `gfold` is the heterogeneous equivalent of a `fold` over a homogeneous container. The implementation of `gfold` can be expressed in terms of two straightforward cases, one for `Insert` and one for `Empty`, following the standard method for folding over a container.

The `Insert` class itself is the metaprogram that makes the structure of a data type available to `gfold`. Its definition relies on *parameterized inheritance*, that is, its own base class is one of the template parameters. This idiom has been widely applied to generating data structures [4, 7], generic containers representing tuples [12], and maps of objects of different types [5, 31]. In our example, the second parameter to `Insert` is used as the base, and the heterogeneous container is built in terms of an inheritance chain (see the appendix for the full class definition). The base class of `Insert` can therefore be thought of as the tail of the list.

#13 C++

```
template <typename Head_, class Tail_>
class Insert : public Tail_
{
    Head_ head_;
protected:
    template <typename Fun_, typename Init_>
    typename Result<...>::Type
    gfold_(Fun_ fun, Init_& init)
    {
        return Tail_::gfold_(fun,fun(init,head_));
    }
public:
    template <typename Fun_, typename InitFun_>
    friend
    typename Result<...>::Type
    gfold(Fun_ fun,InitFun_ initFun,Insert& param)
    {
        return param.gfold_(fun,initFun(param));
    }
};
```

The implementation of `gfold` for such types is split up into two functions. The `gfold` function, a free function defined as a friend, is merely a front-end that creates the respective initial value and passes it to another function, `gfold_`, which performs the fold over data members of the type represented by the given `Insert` instance. The implementation of `gfold_` corresponds exactly to a left fold over a homogeneous list. The call to `Tail_::gfold_` combines the current subterm (represented by the `head_` member) with the initial value and folds the result with the remaining tail. The `gfold_` function for the `Empty` base class just returns the initial value. It is declared `protected`, as `gfold_` is only intended to be called from `gfold` or from the `gfold_` method in a derived class. Note that in contrast to the general case discussed in Sect. 3, `gfold` for `Insert` behaves like `gfoldl`, folding over data members in the order in which they are specified in the definition of the composition.

Providing static heterogeneous containers through metaprogramming and defining algorithms operating on these containers is not a new idea in C++ [31]. The Boost.Fusion library [5] pursues this idea systematically, providing heterogeneous maps, folds, lazy construction by *views*, and more. In particular, it offers return-type deduction for the functions it provides, which allows one to overcome the problems discussed in Sect. 3 and to implement SYB quite directly. We do not use this library for our presentation, as we intend to make explicit the issues of porting SYB to generic C++ rather than hiding them in constructs of de-facto standard libraries.

6. SYB for Classes

Thus far, we have shown that the SYB approach can be implemented in C++ without a loss of genericity and applicability: each feature in the Haskell implementation can be mapped to the corresponding feature of C++ in a quite reasonable way. Yet, the philosophies of the two languages differ. Some assumptions that are natural in Haskell are less likely to apply in C++. In this section, we comment on the differences we observed.

One important difference concerns encapsulation. C++ depends on encapsulation at the level of data types rather than at the level of modules as Haskell. A class rarely exposes its members directly but instead provides a public interface to perform all actions. One reason for encapsulation is that internals of a class in C++ often represent low-level implementation details that should be hidden from the client of the class. The more important reason, however, is that objects encapsulate a state and need to protect this state from being freely modified. Methods like `gfold`, which expose all members, thus break encapsulation in the C++ sense. A class-based application of SYB would therefore use `gfold` most likely for low-level tasks directly related to the data members of a class, for example, for memory-related operations or the export of the state of an object to an external storage format. We note that it would not help to declare `gfold` as a private member function. While the data members then indeed would be invisible, the recursive traversal would at the same time stop to work, since a private method cannot be invoked outside its own class.

As an immediate technical consequence of encapsulation, one is prevented from providing a specialization of `gfold` after completing the definition of a class with private members, which is particularly a problem for classes in legacy code one wishes to adopt. For example, the definition of `gfold` for the `Employee` class in Sect. 4 had to be defined as a `friend` to access the private data members; such friend declarations can only be provided when a class is defined and cannot be retrofitted later.

Classes in C++ are characterized by a separation of interface and implementation. Many classes are even considered *abstract data types* (ADTs), which are types that provide a set of values and associated operations independently from the implementation, by a public interface. Both interfaces and ADTs indicate that one might want to traverse a class in more than one way. The `gfold` function, as discussed so far, always treats a class as a collection of members. To additionally support the interface or ADT view, a class could provide a high-level view of itself for the purpose of folding. In such *interface traversal*, the class designer controls over which subterms `gfold` should fold. For example, SYB could be enabled to treat the Standard Template Library (STL) containers [28] differently from other types by accessing the members on the interface level, by the means of the iterator's interface. In general, interface traversal could be based on concepts [9,26], where a class would be subject of a specific interface traversal if it implements a concept that provides the necessary interface.

Finally, the SYB approach depends on the explicit representation of data; data not explicitly represented but only computed, is not taken into account during traversal. In illustration, imagine that the "paradise" example company has a group of employees with a different kind of contract, which specifies their salary implicitly as a function of the length of their tenure—those employees would not benefit from the salary increase implemented in terms of SYB. The separation of interface and implementation makes it transparent to the clients whether, say, a salary exists as a data member or rather is the result of a method, while the SYB approach forces class designers to both hard-wire and expose their decision. The conflict between SYB and data abstraction is not unique to C++ [22], but the issue is more relevant there, since the explicit structure is not even available for standard lists in C++.

7. Generalization and Extension

In its fully-developed form [16–18], the SYB pattern is more than just the three combinators presented so far. In this section, we discuss some of the issues involved in implementing the remaining features within C++.

We start by discussing how to extend a query or transformation with a new type-specific case, which we anticipate to be straightforward to implement. We continue with the issue of parallel traversal, and note that it would require significant changes to the whole implementation of SYB in C++. Finally, we discuss the implementation of generic queries, which pose interesting questions regarding the genericity of the traversal combinators.

We intentionally focus on algorithms consuming data structures; the implementation of SYB-style producer-algorithms in C++ remains future work.

7.1 Type-Case Extensions

In the SYB approach, type-specific functions have to be made polymorphic. Sometimes, such "extended" functions have to be extended by new type-specific cases. The task of extending polymorphic functions with new type-specific cases applies equally to queries and transformations; we use query extension in our discussion after the second SYB paper [16]. In Haskell, queries are extended by the function `extQ`, which takes a generic query and extends it with a new type-specific case. The implementation of `extQ` has several shortcomings [17], most of them boil down to the fact that `extQ` relies on a dynamic cast operation. The most significant issue, however, is that a practical default case cannot be defined without resorting to mutual recursion and therefore losing extensibility.

The extensibility of `extQ` becomes problematic since the functions in the mutual recursion rely on calling each other by name. Therefore, one cannot simply change one without changing the other. The mutual recursion problem is resolved in Haskell by using type-classes since they enable a function to be overloaded with a type-specific meaning instead of having to introduce a new name. The same solution can be applied in C++; we have already seen overloaded functions at work in the definition of the type-extension function `mkT` in Sect. 3.3.

7.2 Parallel traversals

Within the Haskell implementation of SYB, parallel traversal is defined in terms of zip-like functions, which traverse the elements of two heterogeneous containers in parallel [16]. Since the implementation of `gfold` corresponds to an *internal* iteration of the elements to be folded, such zip-like function cannot easily be defined in a non-lazy programming language [15]. Zipping two sequences requires stepping through the sequences one element at a time. Since `gfold` applies the supplied function to all subterms in one go, there is no easy way to advance one subterm at a time. Even with support for laziness, the iteration would still be complicated since side-effects would also have to be applied in a zip-like manner.

Traditionally, C++ iteration is implemented in terms of *external* iterators, where the caller decides how the traversal proceeds. Implementing SYB in terms of external iterators would however affect almost the entire implementation. In Sect. 5, we discussed the Boost.Fusion library, which implements iterators for traversal of *heterogeneous containers* [5], and recent activities towards support for iterators of hierarchies as required for SYB [21].

7.3 Generic queries

The major remaining part of the SYB approach is the implementation of generic queries and generic query combinators. In Haskell, these are represented by the two functions `gmapQ`, corresponding to `gmapT`, and `everything`, corresponding to `everywhere`. Implementing either of these constructs in C++ does not constitute a fundamental problem, but the implementation would be different because of the lack of laziness. The Haskell implementation of `gmapQ` applies a function to each of the subterms of a given term, and then collects the output in a list. For performance reasons, constructing a list of values to be folded is not an option in C++. Instead, the subterms would be folded directly.

Reasoning about the implementation of `everything` however raises the question whether the genericity of the recursive traversal combinators could be increased. Currently, the two combinators `everywhere` and `everything` construct the same kind of hierarchical traversal, but by using different mapping functions, namely `gmapT` and `gmapQ`.

Can we devise a more generic construct where `everywhere` could be obtained by supplying `gmapT` as the one-layer traversal? An obvious strategy is to specify the recursive traversal in terms of `gfold` instead of `gmapT`. Such a construct, however, is complicated and the convenience of `gmapT` and `gmapQ` is lost.

In Sect. 6, we discussed different ways of folding over the same term, and this motivates parameterizing `everything` and `everywhere` by a folding strategy. The introduction of another parameter is an argument in favor of merging the two, into a more general construct, to avoid duplication.

8. Related Work

Work related to ours can be divided into three categories: approaches to recursive traversal, approaches to automation, and C++ techniques for functional-style programming and advanced type computation. We discuss the first two categories from the standpoint of imperative, object-oriented programming; a good overview of work related to the original SYB approach can be found elsewhere [10, 11, 18].

The natural approach to recursive traversal in the object-oriented world is the *visitor* design pattern [8]. The basic visitor pattern has severe limitations. For one, the action to be performed and the traversal strategy are mixed together [23]. This can be remedied to some degree by *visitor combinators* [29, 30], which allow traversal to be encoded separately from actions. Unlike visitor combinators, our approach does not exploit object-oriented features; we explicitly focus on the generic features of C++. Furthermore, visitor combinators as originally presented cannot be easily applied across different class hierarchies while our approach is universally applicable to any type for which `gfold` is defined. The visitor pattern can be generalized to a non-object-oriented setting. The Loki library [1] provides a generic visitor pattern for C++ and the Boost Graph Library (BGL) [27] employs the visitor pattern to implement generic graph algorithms. Still, even when generalized beyond objects, the visitor pattern does not provide full control over traversal. In C++, traversal is usually represented by *iterators* as in the Standard Template Library (STL) [28]. When it comes to generic traversal, STL-style iterators have two shortcomings, namely, they are designed to iterate over a range of homogeneous values and they impose a performance penalty when iterating over tree-like structures. The first issue has partly been addressed in the Boost.Fusion library [5], the second issue with the introduction of *segmented iterators* [2], which can efficiently traverse non-linear ranges. A technique similar to segmented iterators can be applied to heterogeneous containers [21].

Automation of the SYB approach requires explicit type representation or at least means for reflection on the structure of a type. The SYB approach has been encoded in the "spine" view where the structure of a type is represented explicitly [10, 11]. In our approach, an explicit representation of the structure of values rather than types; values are represented as heterogeneous containers. As explained in Sect. 5, the Boost.Fusion library uses a similar technique.

Functional style programming and advanced type computations for C++ are well-explored topics. The Boost library collection [3] provides the function and lambda [13] libraries for functional programming and the result_of library for type computation. These libraries, however, do not provide all features necessary for implementing SYB. The FC++ [19] library provides a powerful collection of techniques for functional programming and an implementation of a large part of the Haskell Standard Prelude [24]; our implementation could hide technical aspects of currying, higher-order functions, and other functional features in C++ by using FC++. Lastly, the C++ language allows, for example, static or dynamic and strict or lazy computations. Techniques for uniform expression of all available types of computation [25] could make our implementation more widely applicable.

9. Conclusions and Future Work

The SYB pattern in Haskell depends on a number of features that are not directly available in C++, most notably rank-2 types, higher-order functions, and polymorphic type extension. As we have shown in this paper, however, SYB can be implemented in C++ in a way that resembles the original Haskell version quite closely. Not surprisingly, the C++ solution depends heavily on the template feature; both class templates and function templates are used. Other relevant features and idioms include function objects, template member functions, and function overloading.

One of the important design decisions for SYB in C++ concerns the distinction between in-place and out-of-place transformation. In Haskell, the `gmapT` function supports out-of-place transformation, while the natural choice in C++ rather is an in-place transformation. As we have seen, with in-place semantics some parameters from the original Haskell signature become trivial in C++. Of course, it is also possible to support transformation with out-of-place semantics.

The formulation of polymorphic return types that depend on the types of input parameters presents an important technical difficulty for SYB in C++. In Haskell, type signatures allow for type checks of those dependent return types much as any other types. In C++, on the other hand, return types that depend on the types of input parameters must be modeled as class templates that represent the return type computation. Although neither conceptually nor technically very difficult to write, these classes mean extra work and impact the readability of code and, famously, of error messages. At the same time, C++ also simplifies matters: the issue of type extension, which makes the introduction of a special type cast construct necessary in the Haskell implementation, can be handled by function overloading and common specialization techniques. An advantage of the C++ solution, finally, is that it can be provided entirely within the current language standard.

Most of the SYB combinators can be provided once and for all, independently of specific types. The basic one-layer traversal combinator `gfold`, however, must be defined once per type. Although the code for `gfold` can be easily generated by a compiler extension, it nevertheless remains a burden. The need for an external tool can be avoided, granted an explicit representation of types is available within the implementation language. Given such representation, a metaprogram can be used to generate the definition of `gfold`. In Sect. 5, we proposed a particular solution to generating `gfold` and discussed corresponding design issues.

While implementing SYB in C++ we observed that some assumptions, natural in a pure, functional language, are less likely to apply in an imperative language with classes. In particular, SYB views a type as a publicly available collection of its elements. Yet, a class type in C++ often contains internal, low-level representation details that must be hidden. Furthermore, it is natural that a class type provides a public interface to its functionality, independently of the implementation. Consequently, we proposed *interface traversal* for SYB where one-layer traversal can be performed in terms of the one-layer traversal interface that a class provides, rather than directly traversing its members. Finally, SYB assumes that all interesting properties of a type are hard-wired and exposed as one of the explicit constituents, be it a class member or an argument of type constructor application.

In Sect. 7 we discussed possible extensions and generalizations of our implementation outlining the directions for future work. Currently, our implementation does not fully cover SYB functionality. Particularly, we consider implementing query combinators, parallel traversal, and investigating possible generalizations of SYB traversal schemes. Generic producer algorithms [16] constitute an important direction of future work that we have not investigated in this paper. Finally, an extension of our approach to dynamic analysis and traversal of data structures seems natural and promises to be useful for modeling large systems.

Acknowledgments

This work was in part supported by Vetenskapsrådet as well as the project CEDES, which is funded within the Intelligent Vehicle Safety Systems (IVSS) program, a joint research program by the Swedish industry and government. We thank the reviewers of WGP for pointers to related work and suggestions that helped improving the presentation of the paper.

References

[1] A. Alexandrescu. *Modern C++ Design*. C++ In–Depth Series. Addison-Wesley, 2001.

[2] M. H. Austern. Segmented iterators and hierarchical algorithms. In M. Jazayeri, R. Loos, and D. R. Musser, editors, *Selected Papers from the International Seminar on Generic Programming*, pages 80–90. Springer-Verlag, 2000.

[3] The Boost initiative for free peer-reviewed portable C++ source libraries. `http://www.boost.org`.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[5] J. de Guzman and D. Marsden. Fusion library homepage. `http://spirit.sourceforge.net/dl_more/fusion_v2/libs/fusion/doc/html`.

[6] P. Dimov. Boost Bind library. `http://www.boost.org/libs/bind/bind.html`.

[7] U. W. Eisenecker, F. Blinn, and K. Czarnecki. A solution to the constructor-problem of mixin-based programming in C++. In *1st Workshop on C++ Template Programming*, Oct. 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] D. Gregor, J. Järvi, J. G. Siek, A. Lumsdaine, G. D. Reis, and B. Stroustrup. Concepts: First-class language support for generic programming. In *Proc. 21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2006.

[10] R. Hinze and A. Löh. "Scrap your boilerplate" revolutions. In T. Uustalu, editor, *Proc. 8th International Conf. on the Mathematics of Program Construction (MPC)*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[11] R. Hinze, A. Löh, and B. C. d. S. Oliveira. "Scrap your boilerplate" reloaded. In P. Wadler and M. Hagiya, editors, *Proc. 8th International Symposium on Functional and Logic Programming (FLOPS)*, pages 13–29, 2006.

[12] J. Järvi and G. Powell. Boost Tuple Library (BTL) homepage. `http://www.boost.org/libs/tuple/doc/tuple_users_guide.html`.

[13] J. Järvi, G. Powell, and A. Lumsdaine. The Lambda Library: unnamed functions in C++. *Software Practice and Experience*, 33:259–291, 2003.

[14] J. Järvi, J. Willcock, and A. Lumsdaine. Concept-controlled polymorphism. In F. Pfennig and Y. Smaragdakis, editors, *Proc. 2nd International Conf. on Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science, pages 228–244. Springer-Verlag, 2003.

[15] T. Kühne. Internal iteration externalized. In R. Guerraoui, editor, *ECCOP'99 - Object-Oriented Programming, 13th European Conf (ECOOP)*, volume 1628 of *Lecture Notes in Computer Science*, pages 329–350. Springer-Verlag, 1999.

[16] R. Lämmel and S. L. P. Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In C. Okasaki and K. Fisher, editors, *Proc. 9th ACM SIGPLAN International Conf. on Functional Programming (ICFP)*, pages 244–255. ACM Press, 2004.

[17] R. Lämmel and S. L. P. Jones. Scrap your boilerplate with class: extensible generic functions. In O. Danvy and B. C. Pierce, editors, *Proc. 10th ACM SIGPLAN International Conf. on Functional Programming (ICFP)*, pages 204–215. ACM Press, Sept. 2005.

[18] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proc. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI).

[19] B. McNamara and Y. Smaragdakis. Functional programming with the FC++ library. *J. Functional Programming*, 14:429–472, 2004.

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

[21] E. Niebler. Segmented Fusion - a-ha! `http://article.gmane.org/gmane.comp.parsers.spirit.devel/2765`.

[22] P. Nogueira Iglesias. *Polytypic Functional Programming and Data Abstraction*. PhD thesis, School of Comp. Science and Information Technology, The University of Nottingham, UK, Jan. 2006.

[23] J. Palsberg and C. Jay. The essence of the visitor pattern. In *Proc. 22nd Comp. Software and Applications Conf. (COMPSAC)*, pages 9–15, 1998.

[24] S. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. Technical report, Feb. 1999. `http://haskell.org/onlinereport`.

[25] A. Priesnitz and S. Schupp. From generic invocations to generic implementations. In *6th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC)*, July 2006. to appear.

[26] G. D. Reis and B. Stroustrup. Specifying C++ concepts. In *Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 295–308. ACM Press, 2006.

[27] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[28] A. A. Stepanov and M. Lee. *The Standard Template Library*. Hewlett Packard Laboratories, Feb. 1995.

[29] A. van Deursen and J. Visser. Source model analysis using the JJTraveler visitor combinator framework. *Software Practice and Experience*, 34(14):1345–1379, 2004.

[30] J. Visser. Visitor combination and traversal control. In *Proc. 16th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 270–282, 2001.

[31] E. Winch. Heterogeneous lists of named objects. In *Second Workshop on C++ Template Programming*, Oct. 2001.

A. Complete example

```
#include <iostream>
#include <list>
#include <numeric>

// Helper traits
//  to determine function result and parameter types
//  limited to traits required by the example
template <typename Fun_>
struct Result {
   typedef typename Fun_::Result Type;
};
template <typename Fun_>
struct Param1 {
   typedef typename Fun_::Param1 Type;
};
template <typename Result_, typename P1_, typename P2_>
struct Param1<Result_ (*)(P1_,P2_)> {
   typedef P1_ Type;
};

// Implementation of gfold
//  base case followed by standard list example
template <typename Fun_, typename InitFun_,
         typename Param_>
typename Result<InitFun_>::Type
gfold(Fun_ fun, InitFun_ initFun, Param_& param) {
   return initFun(param);
}

template <typename Fun_, typename InitFun_, typename T>
typename Result<InitFun_>::Type
gfold(Fun_ fun, InitFun_ initFun, std::list<T>& param) {
   return std::accumulate<
          typename std::list<T>::iterator,
          typename Result<InitFun_>::Type>(
      param.begin(), param.end(), param, fun);
}

// Implementation of gmapT
//  including required helper function objects
template <typename Fun_, typename Param1_>
struct ApplyToSecond {
   typedef Param1_ Result;
   typedef Param1_ Param1;
   Fun_ fun;
   ApplyToSecond(Fun_ fun) : fun(fun) {}
   template <typename Param2_>
   Param1_ operator()(Param1_ p1, Param2_& p2) {
      fun(p2);
      return p1;
   }
};
template <typename Bound_>
struct Bind {
   typedef Bound_ Result;
   Bound_ bound;
   Bind(Bound_ bound) : bound(bound) {}
   template <typename Param_>
   Result operator()(Param_&) {
      return bound;
   }
};
template <typename Fun_, typename Param_>
Param_& gmapT(Fun_ fun, Param_& param) {
   return gfold(ApplyToSecond<Fun_,Param_&>(fun),
             Bind<Param_&>(param),
             param);
}

// Implementation of everywhere
//  function object followed by convenience wrapper
template <typename Fun_>
struct Everywhere {
   Fun_ fun;
   Everywhere(Fun_ fun) : fun(fun) {}
   template <typename Param_>
   Param_& operator()(Param_& param) {
      return fun(gmapT(*this,param));
   }
};
template <typename Fun_, typename Param_>
Param_& everywhere(Fun_ fun, Param_& param) {
   return (Everywhere<Fun_>(fun))(param); }
```

```
// Implementation of MkT
// function object followed by convenience wrapper
template <typename Fun_>
struct MkT {
    Fun_ fun_;
    MkT(Fun_ fun) : fun_(fun) {}
    template <typename Param_>
    Param_& operator()(Param_& param) {
        return param;
    }
    typename Param1<Fun_>::Type
    operator()(typename Param1<Fun_>::Type param) {
        return fun_(param);
    }
};

template <typename Fun_>
MkT<Fun_> mkT(Fun_ fun) { return MkT<Fun_>(fun); }

// Implementation of explicit data-types
// Class At for accessing the N:th element of a map
// The Empty base case and the Insert cons constructor
template <unsigned int N_, class Map_> class At;

struct Empty {
    template <typename Fun_, typename Init_>
    typename Result<Fun_>::Type
    gfold_(Fun_ fun, Init_& init) { return init; }
};

template <typename Data_, class Base_>
class Insert : public Base_ {
    Data_ data_;
public:
    Insert(Data_ data) : Base_(), data_(data) {}
    template <typename D1_, typename D2_>
    Insert(D1_ d1, D2_ d2)
      : Base_(d2), data_(d1) {}
    template <typename D1_, typename D2_, typename D3_>
    Insert(D1_ d1, D2_ d2, D3_ d3)
      : Base_(d2, d3), data_(d1) {}
protected:
    template <typename Fun_, typename Init_>
    typename Result<Fun_>::Type
    gfold_(Fun_ fun, Init_& init) {
        return Base_::gfold_(fun,fun(init,data_));
    }
public:
    template <typename Fun_, typename InitFun_>
    friend typename Result<Fun_>::Type
    gfold(Fun_ fun, InitFun_ initFun, Insert& param) {
        return param.gfold_(fun,initFun(param));
    }
    friend typename At<1, Insert>::Result
    At<1, Insert>::at(Insert& i);
};

template <unsigned int N_, class Data_, class Base_>
struct At <N_, Insert<Data_, Base_> >
  : At<N_ - 1, Base_> {};
template <class Data_, class Base_>
struct At <1, Insert<Data_, Base_> > {
    typedef Data_ & Result;
    static Result at(Insert<Data_, Base_> & i) {
        return i.data_;
    }
};
template <unsigned int N_, class Map_>
typename At<N_, Map_>::Result at(Map_& m) {
    return At<N_, Map_>::at(m);
}
```

```
// The C++ type hierarchy
// DeptUnit omitted for simplicity
typedef const char *Name, *Address;
typedef Insert<float,Empty> Salary;
typedef Insert<Name,Insert<Address,Empty> > Person;
typedef Insert<Person,Insert<Salary,Empty> >
    Employee, Manager;
typedef Insert<Employee,Empty> PersonUnit, SubUnit;
typedef Insert<Name,Insert<Manager,
    Insert<std::list<SubUnit>,Empty> > > Dept;
typedef Insert<std::list<Dept>,Empty> Company;

// Implementation of currying in C++
template <typename Fun_>
struct Curry;
template <typename Result_, typename P1_, typename P2_>
struct Curry<Result_ (*)(P1_,P2_)> {
    typedef Result_ Result;
    typedef P2_ Param1;
    Result_ (*fun)(P1_,P2_);
    P1_ value;
    Curry(Result_ (*fun)(P1_,P2_), P1_ value)
      : fun(fun), value(value) {}
    Result operator()(Param1 param1) const {
        return fun(value, param1);
    }
};

// Implementation of incS
// binary function followed by curried unary function
Salary& incS(float k, Salary& salary) {
    at<1>(salary) *= 1+k;
    return salary;
}
Curry<Salary& (*)(float,Salary&)> incS(float k) {
    return Curry<Salary& (*)(float,Salary&)>(incS,k);
}

// Implementation of increase algorithm
Company& increase(float k, Company& company) {
    return everywhere(mkT(incS(k)), company);
}

// Implementation of main program
// helper function for list insertion followed by
// a main procedure constructing the "paradise"
// example hierarchy and printing Blair's salary
// after a 20% salary increase

template <typename T>
std::list<T> append(T t, std::list<T> list) {
    list.push_back(t);
    return list;
}

int main() {
    Company company(append(Dept("Research", Employee(
        Person("Ralf", "Amsterdam"), Salary(8000)),
        append(PersonUnit(Employee(Person("Joost",
        "Amsterdam"), Salary(1000))), append(PersonUnit(
        Employee(Person("Marlow", "Cambridge"), Salary(
        2000))),std::list<SubUnit>()))), append(Dept(
        "Strategy", Employee(Person("Blair", "London"),
        Salary(100000)), std::list<SubUnit>()),
        std::list<Dept>())));

    increase(0.20f, company);

    std::cout << at<1>(at<2>(at<2>(
        at<1>(company).front()))) << std::endl;
}
```