A PATTERN FOR STATIC REFLECTION ON FIELDS
Sharing Internal Representations in Indexed Family Containers

Andreas P. Priesnitz and Sibylle Schupp
Department of Computer Science & Engineering, Chalmers Technical University, Göteborg, Sweden

Keywords: Static Reflection, Serialization, Container, Generative Programming, High Performance.

Abstract: Reflection allows defining generic operations in terms of the constituents of objects. These definitions incur overhead if reflection takes place at run time, which is the common case in popular languages. If performance matters, some compile-time means of reflection is desired to obviate that penalty. Furthermore, the information provided by static reflection can be utilised for class generation, e.g., to optimize internal representation.
We demonstrate how to provide static reflection on class field properties by means of generic components in an OO language with static meta-programming facilities. Surprisingly, a major part of the solution is not specific to the particular task of providing reflection. We define the internal representation of classes by a reworked implementation of a generic container that models the concept of a statically indexed family. The proposed features of this implementation are also beneficial to its use as a common container.

1 INTRODUCTION

Reflection provides access to internal representation details of the class of some given object. We consider reflection on the fields of objects, which is essential for generic implementations of low-level tasks. For instance, we are interested in tasks like serializations and optimizing transformations. Many applications of that kind are performance-critical, whereas reflection usually implies some overhead if it is performed at run time. Our goal is to be able to implement those tasks both generically and efficiently.

To avoid performance penalties, we want to apply reflection statically. Common languages do not provide corresponding means. But as some languages offer static meta-programming facilities, we can specify generic components that achieve the desired effect, thus serve as a portable language extension.

Our design relies on sharing the provision of fields for class implementations in a generic component. This component is supposed to manage a collection of objects of different types, thus it forms a heterogeneous container. We want to access those objects in terms of some label, thus the container must model the concept of an indexed family. By separating the aspects of internal class representation, the container becomes a stand-alone generic component, and the task-specific part of our design turns out rather lightweight. Existing implementations of indexed families do not provide all features necessary to use them as internal representations of class fields. Thus, we present an enhanced solution that generalizes the applicability of the container.

The outline of this article is as follows: In Sect. 2, we discuss and specify the applications of reflection that we are particularly interested in. In Sect. 3, we lay out the principal components that constitute our solution. Their heart is the statically indexed family container, whose implementation is in the center of the further discussion. In Sect. 4, we present in general terms the conventional implementation. In Sect. 5, we motivate and describe in more detail necessary extensions to that approach. In Sect. 6, we put the parts together, we show how to ease this step, and we discuss some consequences of that design. In Sect. 7, we compare our approach with previous work in related domains, and in Sect. 8, we give an evaluation of our results and some concluding remarks.

We chose C++ as language of our implementation and of our examples and discuss this choice in Sect. 5.

P. Priesnitz A. and Schupp S. (2007).
A PATTERN FOR STATIC REFLECTION ON FIELDS - Sharing Internal Representations in Indexed Family Containers.
In *Proceedings of the Second International Conference on Software and Data Technologies - PL/DPS/KE/WsMUSE*, pages 30-37
DOI: 10.5220/0001336300300037

2 WHY STATIC REFLECTION?

The term *reflection* encompasses all kinds of access to software meta-information from within the software itself. We restrict our considerations to accessing the instance fields of a given object. Furthermore, we exclude *alteration*, i.e., the modification of that set of fields. In this section, we exemplify two classes of applications that justify to perform reflection statically.

2.1 Example: Serialization

Algorithms whose effect on an object reduces to inspecting its instance fields limit the role of the object to that of a mere record. In other words, these algorithms do not involve knowledge of the particular semantics of the inspected object. A popular example of such applications is *serialization*, i.e., a mapping of objects to some linear representation. Common cases of such representations are binary encodings, XML objects, or formatted output.

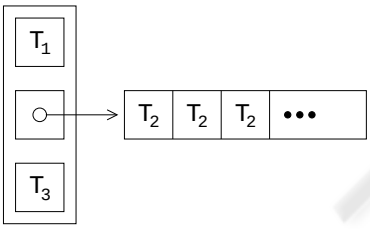

Figure 1: Exemplary internal representation.

In Figure 1, we depict an object that contains

- a reference to a collection of objects of type T_2,
- information on the number of objects in the collection as an object of type T_1, and
- associated information as an object of type T_3.

Consider, for instance, the collection to be a subrange of a text of characters, not owned by the object, and the associated information to be the most frequent character in that text snippet.

We might think of a serialization of that object as resulting in a representation as in Figure 2. Each field is transformed to its corresponding representation, as symbolized by dashed lines, and these individual representations are concatenated. The meaning of concatenation depends on the representation.

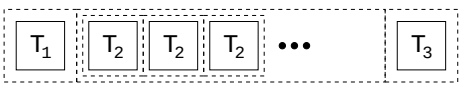

Figure 2: Schematic serial representation.

But a second thought reveals that this example is rather a counter-example: In order to serialize the text snippet, we need to know its length, thus we have to inspect the first field to deal with the second, and we rely on a semantical linkage between both. Thus, a "true" serialization could either create a representation of the reference as such, or treat it as being transparent and represent the (first) referenced object. To produce the outcome in Figure 2, though, we have to specialize the serialization for this type of object.

Some languages provide implementations of certain serialization methods. To define other methods, we need to be able to traverse the fields of objects. Considering common application cases and our previous observation on the example, we restrict these traversals to be

linear: Each field is mapped at most once. The output of the mapping is valid input to a stream.

reversible: Each field is mapped at least once. It is possible to create representations that can be *deserialized* to an object that is equal to the original.

order-independent: The operation does not depend on a particular order of field access. The contrary would require to consider composition information, which we want to exclude.

deterministic: The access order is not random, though. Otherwise, we were forced to add annotations to element representations to facilitate deserialization.

If we lacks means of reflection, we have to define field traversal for each class of interest—in practice, to the majority or entirety of classes.

Like other polymorphic functionalities, reflection usually is a dynamic feature. It causes constant per-object space overhead due to a hidden reference, and a per-reflection time overhead due to an indirect function call. The latter costs are dominated by the impossibility of inlining such calls, and of then applying further optimizations. These costs are insignificant if objects are large and if reflection occurs seldom. Otherwise, the costs of algorithms that make heavy use of reflection may turn out prohibitive, forcing developers again to provide per-class specializations.

This dilemma is avoided if reflection is performed at compile time, abolishing run-time costs in time and space. Run-time reflection is provided by many object-oriented programming languages, e.g., SmallTalk, CLOS, or Java, whereas compile-time reflection is supported only for particular tasks like *aspect-oriented programming*, if at all.

2.2 Example: Memory Layout

Another kind of operations on the fields of a class are optimizing transformations. Usually, only the compiler is responsible for performing optimizations of the internal representation of objects, in terms of space, place, alignment, or accessibility of their allocation. Given adequate means of static reflection and of static meta-programming, one can perform similar transformations within the code. In contrast to the compiler, the developer usually has some high-level knowledge about objects beyond the information expressible in terms of the language. Optimizations that depend on such knowledge complement and support optimizing facilities of the compiler.

As an example, consider the rule-of-thumb of ordering the fields of a class in memory by decreasing size, given that our language allows us to influence that order. The effect of this rule is depicted in Figure 3 for a 2-byte representation of the number of characters. Fields of less than word size may share a word instead of being aligned at word addresses. If we order the fields in such a way that small fields are adjacent, the compiler is more likely to perform that optimization, reducing the object size in the example from 3 to 2 words.

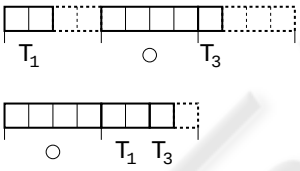

Figure 3: Differences in memory layout.

Note that we do not attempt to be smarter than the compiler: We can not and do not force it to perform that reduction. The compiler may still decide that it is beneficial to stick to word alignment, e.g., in favor of speed of field access, or even to reorder the fields, if allowed. But vice versa, the compiler might not be allowed or clever enough to perform that optimization itself unless the proposed order of fields offers or suggests to do so.[1] Thus, our strategy is to supply the compiler with as much useful information as possible.

Given static reflection, we can inspect the fields of a class and create a new class that contains the fields ordered by size. This effect could not be achieved by a hand-coded alternative, as the size of fields may differ on different hardware. A fixed order that suits some hardware can turn out inappropriate on another.

[1] In fact, GCC has this behavior.

3 COMPONENT-BASED STATIC REFLECTION

We have learned that it is desirable to have static reflection on class fields at our disposal. Thus, we would like to provide this feature ourselves if the language we use does not offer it. In this section, we develop the general outline of our approach to achieve this goal. The individual steps will be discussed in more detail in Sects. 4–6.

3.1 Sharing Internal Representation

In order to share means and applications of static reflection by all class implementations, we provide them within a common generic component. As that component shall be in charge of all matters of (here, static) object composition, it has to be shared by inheritance. Classes have to inherit the component privately to inhibit that unauthorized access is obtained by slicing an object to the component type. To grant public access to functionalities provided by the component, we therefore have to redefine them in class definitions.

3.2 Field Access

Access to fields of an object corresponds to an injective mapping from identifying labels to the fields. To allow for evaluating that mapping at compile-time, reflection support has to be provided by a component that models a *statically indexed family* of fields of different types. Because we required order-independence in Sect. 2, that container does not model a tuple. Due to the expected deterministic access order, though, we require that the container is an ordered indexed family, where we adopt the order of labels from the definition of the container.

In common statically typed languages, labels are lexical entities on which we cannot operate by type meta-programming. Thus, we express labels by empty classes, which do not serve any other purpose than static identification.

3.3 Field Specification

In order to delegate field definition, creation, deletion, access, and further fundamental methods to the family component, we have to statically parameterize it by a collection of field specifications that supply the necessary information. Besides the obviously needed labels and types, one has to specify:

- how to *initialize* fields, depending on the kind of constructor called for the component, and

- how to *delete* field contents in the destructor or when overwriting fields by assignment.

The family has to be self-contained, i.e., able to effect those functionalities for any kind of field types without further support. At the same time, it may not be restrictive; the class developer has to be able to make a choice wherever alternatives of execution exist.

3.4 Separation of Concerns

Not all aspects of reflection on fields can be covered by a general implementation of an indexed family. In order to allow for stand-alone usage of the container, classes do not inherit from the family directly. Instead, a wrapper called `Record` is inherited, which complements the container features by particular support for reflection. In general, that wrapper serves as a place to specify features that apply to all objects.

Figure 4 depicts the inheritance relationship between the essential components in our design.

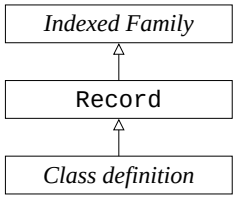

Figure 4: Principal component hierarchy.

4 IMPLEMENTING HETEROGENEOUS FAMILIES

Statically polymorphic containers store objects of different types. They are common in generic models and provided by several, mostly functional languages. We describe an established approach to defining such containers by user-defined constructs. Our language has to allow expressing *generative* constructs, as different container types must be created for different sets of element types. *Java*, for example, is not suitable: It performs *type erasure* on instances of generic types, instead of creating new types. More generally, our language has to allow conditional type generation, i.e., *static meta-programming* (*SMP*). Candidates are MetaML, MetaOCaml, Template Haskell, or C++.

Compile-Time Lists. Given SMP, we can define lists of types by a binarily parameterized type `Cons` and a type `Nil` in the same style as lists are defined in LISP (Czarnecki and Eisenecker, 2000). Similarly, a list algorithm is implemented as a type that is parameterized by a type list and that provides the result as an associated type.

Mixins. In order to influence class composition, our language has to allow *parameterized inheritance*, i.e., inheriting from a base class that is given as a type parameter. The subclass is then called a *mixin* (Bracha and Cook, 1990) and generically extends the base class. We are interested in mixins that add a single field and corresponding functionalities to the base class. A particular problem is to provide practicable mixin constructors that call the base class constructors properly (Eisenecker et al., 2000).

Tuples. Combining the previous idioms, tuples of objects of different types can be defined recursively (Järvi, 1999; Järvi, 2001): `Cons` becomes a mixin that inherits the list tail and that provides a field of the head type. `Nil` is an empty class.

Elements are accessed by a static natural index. If the number is nonzero, the mixin forwards the access request with the index' precessor statically to the base. If it is zero, the field in the mixin is returned. This static recursion causes no run-time overhead.

The tuple class implementation is straightforward. But its constructors are tricky to define, as they need to handle arbitrary numbers[2] of initializing arguments. Furthermore, the creation of temporaries has to be avoided when passing arguments on from a mixin constructor to a base constructor.

Families. An arbitrarily, yet statically indexed family is implemented analogously to a tuple (Winch, 2001; Weiss and Simonis, 2003): The mixin class is parameterized by a list—assumed to represent a set—of type pairs `Entry<Label,Type>`. The second type indicates a field type, whereas the first type is an identifying label to be matched, instead of an index, when accessing the field. The list of field specifiers does not necessarily determine the order in which the mixins are composed. Thus, we may reorder that list before creating the family, e.g., in decreasing order of field type size.

Constructor implementations differ from those for a tuple, but are similarly intricate. We are not tied anymore to a certain order of arguments nor forced to provide all of them. Instead, constructors take as initializers a set of label-value pairs that need to be searched for a matching label.

Figure 5 illustrates the general layout of our proposed family implementation. In the following, we

[2] possibly up to a sufficiently large limit

will justify the individual components and discuss their relevant features.

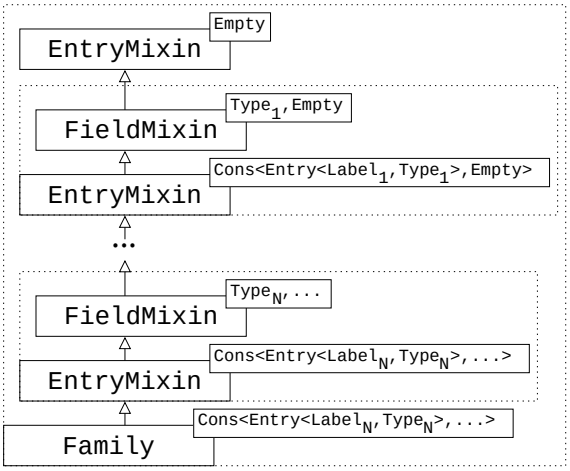

Figure 5: Family implementation.

5 INCREASING GENERICITY

The semantics of statically indexed families have to be refined in order to serve the internal representation of objects. The principal reason is that any possible field type has to be supported, even types that hardly occur as container element types. Moreover, for each of the fields individual behavior, e.g., that of constructors, may have to be specified.

In this section, we discuss how to deal with different aspects of those problems. We avoid discussing language-specific details and focus on conceptual issues. Our actual implementation, though, was effected in C++. Of the languages that allow static metaprogramming (see Sect. 4), C++ is the most spread and the most capable of modeling low-level performance-critical details.

5.1 Arbitrary Kinds of Element

To allow for any type of elements in the family, we have to provide support for even those types that are not or only partially covered by common implementations of generic containers, e.g., arrays or references in C++. Therefore, each mixin in the original design is replaced by a pair of mixins: We extract the actual field definition from the mixin class to another mixin class that serves as its base. In the remainder of the original mixin we dispatch upon the field type to choose that base mixin appropriately.

5.2 Initialization

In constructors of a generic container, proper element initialization is tedious. Not only does the initialization depend on the element type, but it may also be necessary to treat particular elements in non-"standard" ways. Furthermore, we need a means to initialize in place, to avoid the creation of temporary objects when delegating initialization by an expression. Therefore, we refer for each family element to:

- a nullary functor returning the initializer to be used by the default constructor,
- a generic unary functor taking a family argument and returning the initializer to be used by copy constructors, and
- a unary functor taking the element as argument and returning no result, to be called by the destructor.

These functors are provided and possibly overwritten in the following order:

1. Type-dependent default implementations are given by the type-specific mixins that provide the fields.
2. For each element of the family, the user may specify such functors in the corresponding entry in the parameter list, additionally to the label and element type.
3. In a constructor call, the functors to use for particular fields can be optionally specified, see below.

A family has the following constructors besides the usual default and copy variant:

- A generic copy constructor takes a family argument. Fields are initialized by the result of applying the corresponding functor to that family.
- An initializing constructor takes as arguments an arbitrary number of pairs of labels and nullary functors. Fields are initialized by the result of the functor paired with the corresponding label.
- A hybrid alternative takes as arguments a family and an arbitrary number of initializing pairs. Here, the second entries of the pairs are unary functors that are applicable to the family.

Assignments are defined by default as sequential field deletion and initialization in terms of those functors. Other field-specific functions, like swapping or cloning, expose the usual type-specific behavior.

5.3 Element Traversal

Access to family elements is provided in the conventional way (see Sect. 4). For traversals over the

elements, we provide combinators like `fold`. But our implementation of `fold` differs from similar approaches (de Guzman and Marsden, 2006) in that we fold over the labels rather than the fields, that are then accessed in terms of the labels. This way, we can express role-specific behavior, e.g., serialization to XML.

According to the requirements in Sect. 2, the traversal order has to be deterministic. Thus, `fold` follows the order defined by the user in the specification list parameter of the family. The actual order of mixin composition is not regarded.

Given `fold`, generic serialization is elegantly expressed along the lines of the *Scrap your Boilerplate* pattern (Lämmel and Peyton Jones, 2003). Recursive traversal over an object is expressed by applying `fold` to the object, performing on each of its fields the same kind of traversal. Only for primitive types the serialization has to be specialized appropriately.

5.4 Reflective Operations

Transformations like the memory optimization in Figure 3 need to be performed once per family type. Therefore, the actual family is represented by a wrapper `Family` that inherits the recursive implementation, see Figure 5. The wrapper may preprocess the parameter list, e.g., reorder it, before passing it on to the mixin hierarchy. According to the previous discussion, traversing algorithms are not influenced by this transformation.

6 COMPONENT ASSEMBLY

We suggested the principle of implementing classes in terms of statically indexed families. Then, we discussed aspects of implementing such families generically. In this section, we put those parts together.

6.1 Sharing Internal Representation

The following example illustrates how to define the text snippet class from Sect. 2. As discussed in Sect. 3, we add fields to the class by inheriting privately from a family instance `Family<...>`. The fields it provides are specified by its parameter, a `Cons`-created list of label-type pairs `Entry<...>`. The labels, like `Number`, are individual types without particular semantics. The constructor takes a string and start/end positions. To initialize the fields, the constructor delegates label-value pairs created by `init` to the constructor of the family base object. The function `most_frequent` detects the most frequent value in the given pointer range.

```
class Snippet
: Family<Cons<Entry<Number,short>,
         Cons<Entry<Text,char const*>,
         Cons<Entry<Most_Frequent,char>,
         Nil> > > >
{
    typedef Family<...> Base_;
public:
    Snippet(char const* text,
            unsigned from, unsigned to)
    : Base_(init(Number(),to-from),
            init(Text(),text+from),
            init(Most_Frequent(),
                 most_frequent(text+from,
                               text+to)))
    {}
    // ...
};
```

We have to overwrite functionalities provided by the family if their semantics differ from the default. For instance, assignment has to be redefined to return the actual object instead of its family base-object. Such boilerplate code has to be provided in each class that makes use of a family as its field container. Hence, we want to share that code in another component.

6.2 The Record Front-End

In order to share functionalities that are specific to classes rather than to families, class definitions inherit a parameterized class `Record` instead of a `Family` instance. `Record` inherits the family and transforms its container semantics into record semantics, making it more than the sum of its parts. For instance, the wrapper provides reflective methods like serialization by applying corresponding algorithms on the family.

A similar purpose of `Record` is to share general features of objects, e.g., automatic generation of an identifier. The implementation of some features may require that class definitions provide additional parameters to `Record`. When introducing such features, we therefore need to upgrade existing code. The use of an explicit construct for composition allows one to use refactoring tools for this task.

In some sense, `Record` models the concept of a common superclass, like `Object` in Java. But objects are not supposed to be used as a `Record` instance. We have to rely on coding discipline in this regard: If `Record` were inherited privately, it could not provide boilerplate methods to the class interface. In fact, we should provide that code in a common class front-end rather than in a base class, requiring a more extensive design of class definitions. For the scope of this discussion, we consider `Record` sufficiently helpful to accept the lack of a formal obstacle to slicing.

6.3 Class Composition

In its current form, our approach relies on a linear inheritance hierarchy and on a closed specification of all fields of a class. We run into trouble when we define a class not monolithically, but by inheriting some other class. That definition bases on (at least) two families whose features are not automatically combined. A straightforward solution is to avoid class composition and to rely exclusively on object composition. In fact, inheritance is hardly used in performance-critical software where the need for static reflection arises.

But class composition from components is feasible, as long as these do not contain field definitions. We see the mission of our approach in role-based and generative design, where classes are assembled from collections of components. The extraction of the respective fields into a family serving as base class is an initial step of class generation. A final step would be the provision of the common class front-end mentioned before. The application to component-based class design was a major motivation behind our requirement of independence on field order in Sect. 2.

7 RELATED WORK

To our knowledge, no previous attempt has been undertaken to provide static reflection on object fields or corresponding effects by means of appropriately designed generic containers.

Several publications under the label *SYB*/*Scrap your Boilerplate* (Lämmel and Peyton Jones, 2003) aim at generically expressing recursive traversals of data structures in Haskell. They rely either on constructs that need to be specialized per data structure, or on appropriate language extensions. The *spine view* approach (Hinze and Löh, 2006) generalizes SYB by mapping algebraic data types to recursively defined types that expose the fields one by one. A fundamental difference to our approach is that Haskell is functional, which makes the exposition of fields uncritical. To ensure encapsulation, our internal representation has to be kept private, and functions like serialization have to be part of the class interface.

A previous study on expressing SYB in C++ (Munkby et al., 2006) incorporated a `fold` definition in terms of a heterogeneous container. That container itself was used as an object, similar in style to Haskell, which led to conflicts between encapsulation and applicability. Our approach avoids such problems by separating a class from its field container and encapsulating both the container and its field traversal functions. Class definitions only expose a high-level interface to methods like serialization.

The fusion library (de Guzman and Marsden, 2006) provides a rich selection of sophisticated implementations of heterogeneous containers of different kind, including families (called `maps`) as well as the corresponding algorithms and adaptors in the style of the C++ STL/Standard Template Library. As for now, there is no sufficient support for the issues addressed in Sect. 5. Hence, these containers can not serve to provide the fields of classes. But the differences are slight, and the combination of fusion constructs with our approach is both feasible and promising.

As discussed in Sect. 6, our work aims at solving common problems in component-based class design. Research in that domain focused on splitting classes horizontally by functionality (Czarnecki and Eisenecker, 2000; Smaragdakis and Batory, 2000) rather than on vertical splits by class structure. The question of forwarding constructor calls through mixin hierarchies has been addressed before (Eisenecker et al., 2000). The solution relies on wrapping constructor arguments in a tuple, obeying the composition order.

Another paper (Attardi and Cisternino, 2001) proposes to add more extensive meta-information to C++ classes by means of preprocessor instructions and meta-programming. On the one hand, field and method definitions have to be given separately and are not created by the same instructions. On the other hand, the meta-information is provided at compile time, but not used for class generation.

Finally, *multi-stage programming* languages like MetaOCaml (Calcagno et al., 2003) incorporate meta-information and code creation in the language itself. Unfortunately, they are not very spread yet. Our approach provides a subset of such features in reusable components. In contrast to language-based features, library-based language extensions are portable, applicable in a popular performance-oriented language like C++, and open to high-level extensions beyond the language's expressiveness. We believe the combination of both approaches the key to mastering complex and computationally expensive software.

8 CONCLUSION

We discussed how to provide generic algorithms that rely on meta-information about their arguments' internal representation, but to avoid the performance penalties of dynamic reflection. Static reflection is a rare language feature, but statically generic programming allows to provide such features in reusable components. We restricted our discussion to linear, order-independent, but order-preserving algorithms, like se-

rializations. Our solution allows to implement generically the serialization and the optimization described in Sect. 2, without imposing run-time overhead. Similarly, any type-dependent operation of the specified kind can be implemented.

We extract field information from class definitions and represent it in containers that model statically indexed families. It is known how to implement such containers, but shortcomings of conventional implementations have to be resolved for our purpose. In particular, support of unusual element types and proper initialization need to be ensured. Linear algorithms are expressed elegantly in terms of higher-order combinators. Optimizations and other features are beneficial for either use of the construct as generic container or as internal class representation.

Class definitions that inherit their fields from a container involve boilerplate code. We propose to share that code in a wrapper that makes the container a record. We accept a conflict with the principle of encapsulation and leave a more complete solution to future work on component-based class composition.

As it relies on static meta-programming, our solution pays run-time efficiency by increased compilation efforts. We consider these costs tolerable for performance-critical systems, and for objects with rather few fields. In other cases, dynamic reflection may be more appropriate.

Our approach attempts to deal with as many aspects and uses of field definition and reflection as possible within the underlying container. But we do not claim or require their complete coverage. A class definition in the proposed style allows to influence or override any behavior of the underlying family. Only essential and invariable properties are encapsulated.

The solution is a flexibly applicable implementation pattern, sufficiently abstract to be expressed entirely in terms of portable library components, and that does not rely on compiler extensions. We can conveniently add extensions to the components, that automatically enrich the semantics of all classes defined by that pattern. We consider these aspects significant advantages of our solution over alternative approaches that rely on language extensions and compiler facilities. These solutions are safer to use, but harder to propagate and extend.

ACKNOWLEDGEMENTS

We are grateful to Gustav Munkby and Marcin Zalewski for inspiring discussions, to the reviewers for useful comments, and to the Swedish Research Council (Vetenskapsrådet) for supporting this work in part.

REFERENCES

Attardi, G. and Cisternino, A. (2001). Reflection Support by Means of Template Metaprogramming. In *Proc. 3rd Int. Conf. on Generative and Component-Based Software Engineering (GCSE)*, volume 2186 of *LNCS*, pages 118–127. Springer.

Bracha, G. and Cook, W. (1990). Mixin-Based Inheritance. In *Proc. 5th ACM Symp. on Object Oriented Programming: Systems, Languages and Applications (OOPSLA)*, volume 25, number 10 of *ACM SIGPLAN Notices*, pages 303–311. ACM Press.

Calcagno, C., Taha, W., Huang, L., and Leroy, X. (2003). Implementing Multi-Stage Languages using ASTs, Gensym, and Reflection. In *Proc. 2nd Int. Conf. on Generative Programming and Component Engineering (GPCE)*, volume 2830 of *LNCS*, pages 57–76. Springer.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools and Applications*. Addison-Wesley.

de Guzman, J. and Marsden, D. (2006). Fusion Library Homepage. `http://spirit.sourceforge.net/dl_more/fusion_v2/libs/fusion`.

Eisenecker, U. W., Blinn, F., and Czarnecki, K. (2000). A Solution to the Constructor-Problem of Mixin-Based Programming in C++. In *Proc. First Workshop on C++ Template Programming at 2nd GCSE*.

Hinze, R. and Löh, A. (2006). "Scrap Your Boilerplate" Revolutions. In Uustalu, T., editor, *Proc. 8th Int. Conf. on Mathematics of Program Construction (MPC)*, volume 4014 of *LNCS*, pages 180–208. Springer.

Järvi, J. (1999). Tuples and Multiple Return Values in C++. Technical Report 249, Turku Centre for Computer Science.

Järvi, J. (2001). Boost Tuple Library Homepage. `http://www.boost.org/libs/tuple`.

Lämmel, R. and Peyton Jones, S. (2003). Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proc. ACM SIGPLAN Int. Workshop on Types in Language Design and Implementation (TLDI)*, volume 38, number 3 of *ACM SIGPLAN Notices*, pages 26–37. ACM Press.

Munkby, G., Priesnitz, A., Schupp, S., and Zalewski, M. (2006). Scrap++: Scrap Your Boilerplate in C++. In *Proc. 2006 ACM SIGPLAN Workshop on Generic Programming (WGP)*, pages 66–75. ACM Press.

Smaragdakis, Y. and Batory, D. (2000). Mixin-Based Programming in C++. In *Proc. 2nd Int. Symp. on Generative and Component-Based Software Engineering (GCSE)*, volume 2177 of *LNCS*, pages 163–177. Springer.

Weiss, R. and Simonis, V. (2003). Storing properties in grouped tagged tuples. In *Proc. 5th Int. Conf. on Perspectives of Systems Informatics (PSI)*, volume 2890 of *LNCS*, pages 22–29. Springer.

Winch, E. (2001). Heterogeneous Lists of Named Objects. In *Proc. Second Workshop on C++ Template Programming at 16th OOPSLA*.