The Translator Pattern —
External Functionality with Homomorphic Mappings

Thomas Kühne
kuehne@informatik.th-darmstadt.de
Department of Computer Science
Darmstadt University of Technology
Magdalenenstr. 11c, D-64289 Darmstadt

Abstract

We describe how to express homomorphic translations with an object-oriented design. This includes a uniform approach to the addition of external functionality to data structures. We present the tradeoffs of distributing intrinsic and extrinsic properties to data structures and external functions respectively. A mini-architecture (design pattern) for the emulation of multi-dispatching functions is presented in order to allow the definition of local translation functions. Actually, we present a variant of the Visitor pattern, but add the idea of maintaining a homomorphism between source and target and introduce a target structure between source structure and target semantics. Combined, these concepts pave the way to incremental evaluation.

1 Introduction

Many operations on data structures can be viewed as homomorphisms, that is, as structure preserving mappings from one domain into another. For instance, compilers typically map the abstract syntax of the source language into a specific machine code language[1]. Other kinds of abstract interpretations (e.g., pretty-printing and type-checking) should be expressed as homomorphisms between source and target domain as well. The reason for this recommendation can be explained by means of an equation that holds, if a homomorphic relationship between two structures exists:

$$\phi(op(a,b)) = op'(\phi(a),\phi(b)) \qquad (1)$$

An interpretation ϕ on an operation op (from a source domain) with subcomponents a and b is defined as a new operation op' (from a target domain) whose subcomponents are determined by again applying ϕ to a and b [Wechler 92]. Applied to compilation this looks like:

```
compile(assign(lhs, rhs)) =
store(compile(lhs), compile(rhs))
```

First of all, the above equation depicts a general way of shifting the interpretation from operators down to operands. The strictly "top-down" recursive nature of this translation process should be recognizable for users of an object-oriented

Appeared in Conference Proceedings of TOOLS USA, Santa Barbara, California, editors Christine Mingins and Bertrand Meyer, pages 48–62, 1997

[1]Historically, homomorphisms are closely connected to syntax-directed translations [Aho 86] and correspond to compositional definitions [Nielson 93].

design. Furthermore, the right hand side of equation 1 has a structure that allows us to account for incremental modifications to the source structure. Consider a change made to the left-hand-side (`lhs`) of `assign`. There is no need to rebuild the whole result term. One simply has to apply `compile` to the changed `lhs` and plug the result into the first operand of `store`.

A homomorphic translation is allowed to perform a non-injective mapping, i.e., different elements of the source domain may be mapped onto the same element in the target domain. For instance, an initialization statement for a variable could be translated to a `store` instruction as well. In the target domain (machine language) there is no way to tell which type in the source domain caused the generation of a `store` (see section 2.7 *Separation* for further homomorphic simplifications).

Subsequently, we describe a general approach for homomorphic translations in the form of a design pattern [Gamma 94]. The Translator pattern joins four aspects:

1. Homomorphic translations
2. Separation of translation and semantics
3. Potential incremental translations
4. External polymorphism.

The last point is of technical nature but will immediately be explained within the core pattern description as it substantially influences the pattern's structure. We already elaborated on point one, i.e., assigning a meaning to a recursive structure in terms of a similarly structured recursive meaning. Point two refers to the fact that this recursive meaning is represented by an explicit intermediate data structure. Thus, the final semantics of the translation is given by a function on the intermediate data structure. This aspect, as well as point three, will be discussed in the pattern's consequences (section 2.7).

2 Design Pattern: Translator

2.1 Intent

Add semantics to structures with heterogeneous elements without changing the elements. Separate interpretations from each other and use local interpretations that allow for incremental reevaluation.

2.2 Motivation

Consider a programming environment that represents programs as abstract syntax trees. It will need to perform various interpretations on the abstract syntax tree like type-checking, code generation, and pretty-printing. Figure 1 depicts two sample transformations.

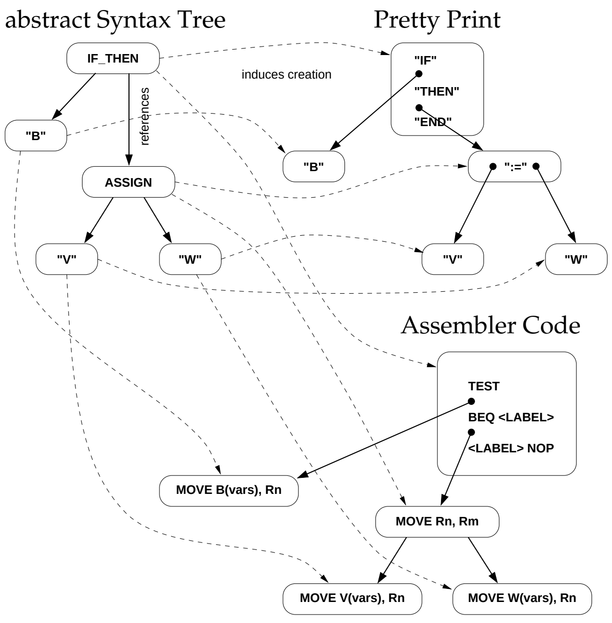

Figure 1: Homomorphic translations of trees

The result of a mapping (dashed arrows in figure 1) depends on the interpretation (e.g., compilation) and concrete node type (e.g., assign) involved. One may put all various interpretations (type-check, pretty-print, etc.) into the node interface in order to rely on dynamic binding. However, this is often not a good idea:

- It leads to a system that is hard to understand, maintain, and change.
- Adding a new interpretation means changing and recompiling all node types.
- An interpretation can not be added without changing the node interface.
- The interface of nodes will grow until it becomes bloated.

The first two arguments are also addressed by the Visitor pattern [Gamma 94]). Visitor also addresses the problem of adding functionality to each node-type (represented by a class) in a conceptual hierarchy (e.g., abstract syntax, construction data, etc.) but does not aim at incrementality and demands node-types to know about external interpretations (see section 2.10).

The last two arguments of the above list especially apply to data structures other than abstract syntax trees. Consider a data structure that represent the logical structure of a building. It is probably only well after designing the interface to that structure that one wishes to perform some interpretation like computing the total rent income. In this context, it is useful to differentiate between intrinsic properties (nodes have descendents) and extrinsic properties (pretty-print). There is no end to extrinsic properties and it does not make sense to lump all of them into one interface.

Now, if we provide interpretations as external features we are facing a problem with an implementation language that provides single-dispatch only[2]. As already mentioned, the code to be executed for each node when we traverse an abstract syntax tree depends on two variabilities:

```
exec-code(node-type, interpretation)
```

Note that we already rejected `node-type.interpretation` with the argumentation above. The reverse, `interpretation.node-type`, does not make sense, since, unlike the interpretation type, the node type always changes during tree traversal; that is, dispatch isn't required for the receiver but for the argument.

What we need is double-dispatch on both `node-type` and `interpretation`. Fortunately, there are ways to emulate double-dispatch and its generalization multi-dispatch, with a single-dispatch language. We opt for a solution which can be characterized as external polymorphism (see section 2.10 for Visitor type double-dispatch). Unlike Cleeland et al., however, we do not use a combination of C++ templates, Adapter, and Decorator [Cleeland 96]. We simply use generic functions [Kühne 96].

When a generic function object is applied to a node, it determines the node's type, creates the corresponding specialized function object, and returns the result of applying the specialized function object to the node.

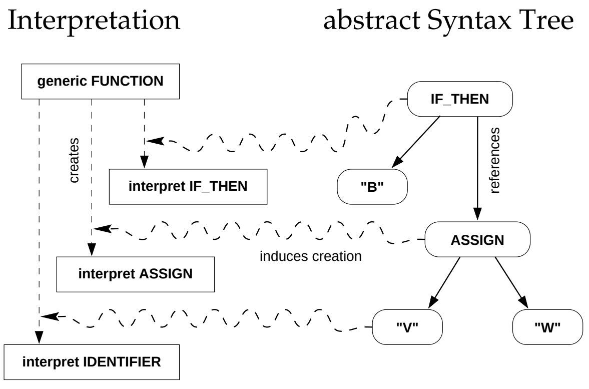

Figure 2: Generic interpretation

Figure 2 depicts how concrete element types (IF_THEN) induce the creation of their corresponding specialized functions. A specialized function knows the exact type of its argument

[2]Languages with multi-dispatch, e.g., CLOS, Cecil, or Dylan are not in widespread use.

and, therefore, can appropriately exploit the argument's full interface.

Note that it is not only natural to deal with generic functions to achieve double-dispatch, but also very natural to employ functions for translations. The approach of formally defining the semantics of a programming language called denotational semantics is entirely based on semantic functions, i.e., functions that transform phrases into denotations [Schmidt 86].

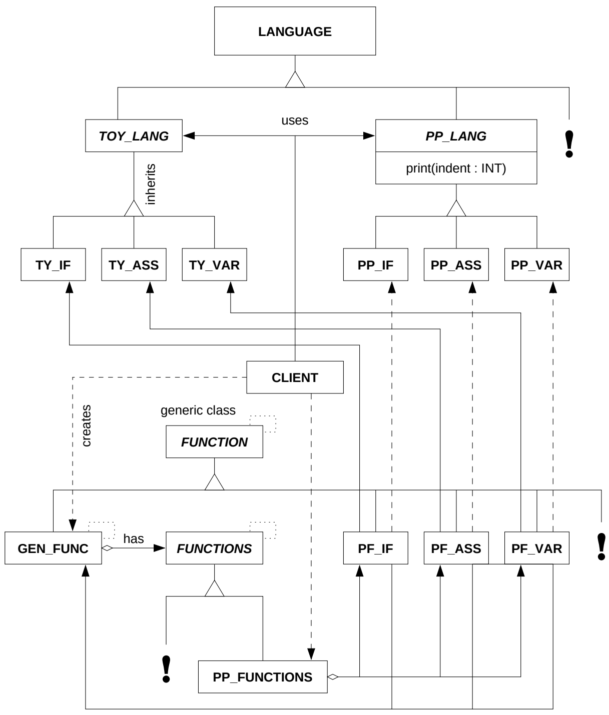

Figure 3: Sample structure

Figure 3 shows the structure diagram that corresponds to the domains used in figure 1. Only relationships relevant to Translator have been included. For instance, language nodes like TY_IF will typically have an aggregation relation with TOY_LANG. Exclamation marks denote places of possible extension (see section 2.7 *Extensibility*).

Class LANGUAGE in figure 3 is not required in general (see figure 4). Also, it is not required that TY_IF, TY_ASS, etc. have a common ancestor (like TOY_LANG). Hence, one can define semantics on heterogeneous collections where element types may come from different libraries.

2.3 Applicability

Use the Translator pattern for

- *Adding semantics.* When you want to add an interpretation (even without having planned for it) to a number of classes that have different interfaces, Translator allows accessing the heterogeneous interfaces individually. The classes need not belong to the same hierarchy or library.

- *External functionality.* Adding interpretations outside of elements avoids bloating the elements' interfaces with extrinsic concepts. Also, if interpretations require additional servers (e.g., environment lookup for type-checking) the *interpretations*, as opposed to the *elements*, will depend on the servers, i.e., require recompilation in case one server changes.

- *Incrementality.* When small changes to big structures should not cause reevaluation of the whole structure, exploit the homomorphic properties of Translator and use the intermediate structure (see figure 6) for storing computed results.

Do not use the Translator pattern in case of

- *Unstable elements.* When new elements are frequently added to the source structure it is probably better to define the interpretations in the elements. Otherwise, one has to constantly change all associated function packages (see figure 4 and also section 2.7).

- *Space constraints.* Unless you can make use of the benefits of a target structure (see section 2.7 *Separation.*), avoid the space overhead by directly translating to results.

2.4 Structure

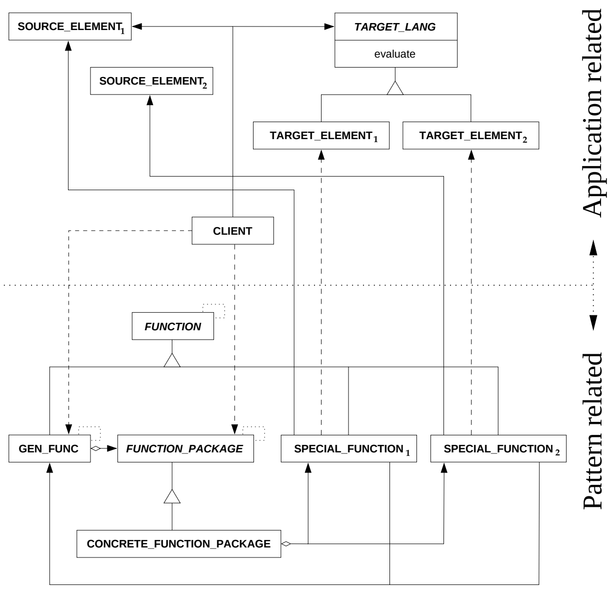

Figure 4: Structure diagram

2.5 Participants

- **Function** (FUNCTION)
  - declares an interface for function application. Its two type parameters specify argument and result type respectively[3].
  - is used as the interface specification for both generic and specialized functions.
- **Generic function** (GEN_FUNC)
  - corresponds to a denotational function definition.
  - uses function package FUNCTIONS and run time type information to choose and then delegate to a specialized function.
- **Specialized function** (e.g., PF_IF)
  - corresponds to one pattern matching branch of a denotational function definition.
  - defines a local transformation for a source element (e.g., TY_IF) to a corresponding target element (e.g., PP_IF).
  - recursively transforms subcomponents of its argument as well.
- **Function package** (e.g., FUNCTIONS)
  - conceptually bundles related specialized functions.
  - declares a generic type for specialized functions to be refined by concrete function packages.
- **Concrete Function package** (e.g., PP_FUNCTIONS)
  - defines a mapping from source elements to their corresponding specialized functions.
  - creates prototypes of — and then aggregates — specialized functions.
- **Client**
  - creates or uses a source structure (e.g., TOY_LANG).
  - initializes or uses a function package (e.g., PP_FUNCTIONS).
  - creates or uses a generic function (GEN_FUNC).
  - applies a generic function to a source structure.

2.6 Collaborations

Figure 5 shows important object interactions. It refers to "@" for an infix function application syntax (see sample code in section 2.9).

[3]As is the case with all generic functions of figure 4.

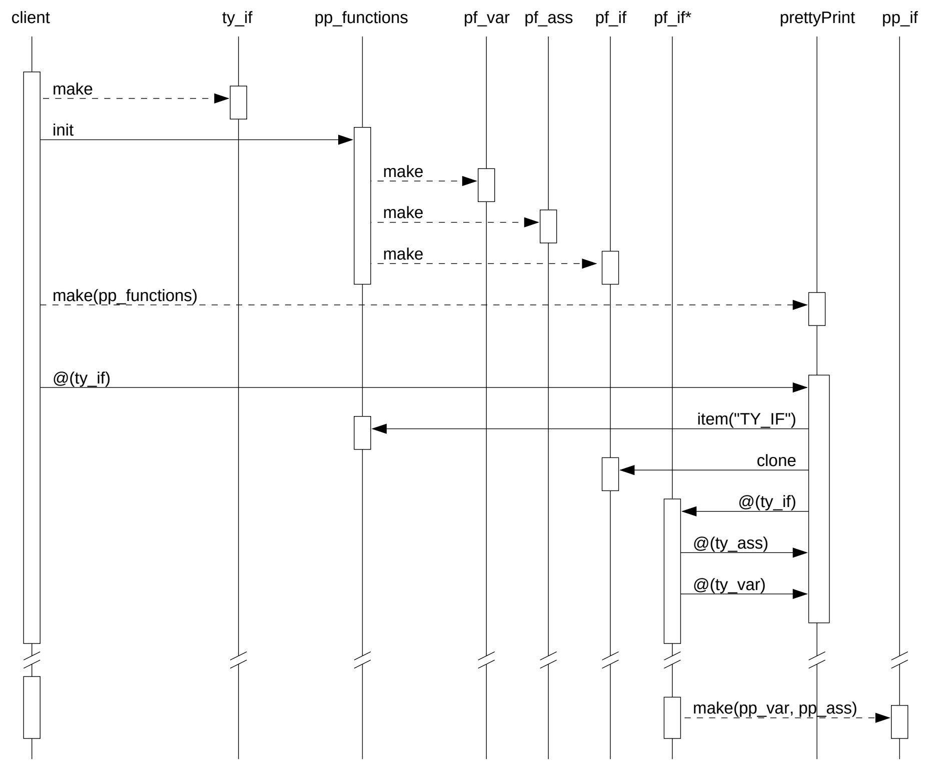

Figure 5: Interaction diagram

- A client initializes a function package in order to create a generic function from it. The client applies the generic function to the source structure in order to obtain the translation result.

- The generic function consults the function package for a specialized function that matches the type of the argument. Then it applies a cloned exemplar of the specialized function to the argument.

- A specialized function recursively calls its associated generic function to the subcomponents of its argument. Then it creates the target element while providing it with the results of the subcomponent evaluation.

Note that the time line gaps in figure 5 denote potential recursive mappings of subcomponents.

2.7 Consequences

Tradeoffs of Translator are:

- *External functionality.* Translator makes it easy to add interpretations to data structures with heterogeneous elements. In contrast to Visitor [Gamma 94] there is no need to impose an `Accept` method on the elements. Spreading interpretations over all elements (i.e., object-oriented design) would demand changing all elements when introducing a new interpretation. Gathering all related behavior into one generic function (i.e., functional design) — thus separating unrelated behavior (e.g., compilation from pretty-printing) — results in a clean partition and allows to hide interpretation specific details (like interpretation specific data structures and accumulated state) in generic functions.

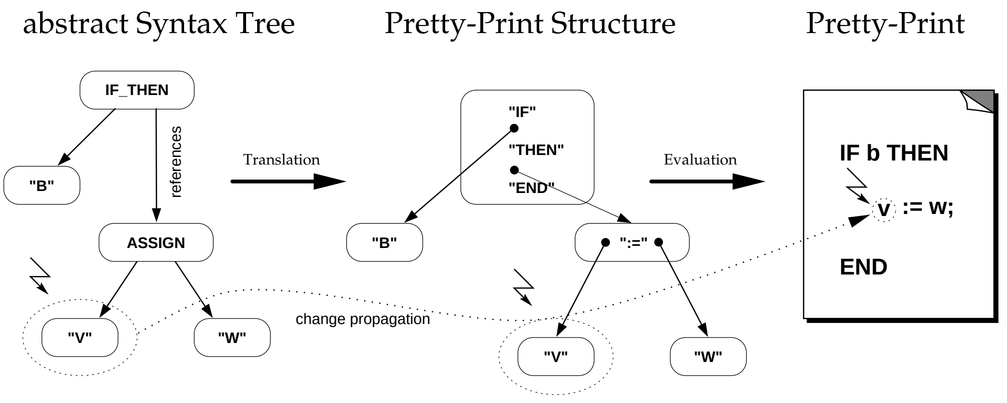

Figure 6: Distinct interpretation phases

- *Instability.* When using Translator, adding new elements (e.g., changing the abstract syntax of a language) becomes difficult. Unlike Visitor, Translator does not demand that one extends all interpretations with a meaning for a new element (e.g., compilation is not affected by adding a new type-declaration node). However, updating of concrete function packages and creation of specific functions is required. Note that the latter point must be done anyway, but if interpretations are element methods, then their completeness can be enforced by the compiler. Using Translator, runtime errors caused by an undefined mapping to a specialized function may occur.

- *Extensibility.* It is easy to add new translations and/or target structures. A new interpretation simply requires

  1. defining a new target structure (top exclamation mark in figure 3),
  2. defining specialized functions (rightmost exclamation mark), and
  3. providing a function package that maps the source elements to their specialized functions (leftmost exclamation mark).

  The last action is a tribute to the emulation of generic functions.

- *Flexibility.* Elements to be translated need not be from one class hierarchy (a counter example is Iterator [Gamma 94]). In any case, the full individual interface can be accessed by the associated specialized functions.

- *Broken encapsulation.* Since specialized functions can access the public interface of elements only, the interface may be forced to be wider than necessary for other clients. A remedy is to use selective export or another kind of “friend” mechanism.

- *Separation.* The semantics of an interpretation is defined in terms of a target structure semantics (see figure 6). Thus, a clear separation between translation (mapping to a target) and target semantics (meaning of target) is achieved. Figure 7 depicts how an *interpretation* is split into a *translation* to a new *Target* structure and an *evaluation* function that produces the final result. During the translation several simplifications are possible (see table 1).

  A pretty-print, therefore, is not a sequence of side effects but, at first, a hierarchical structure of print-elements, combinators,

Notation	*Meaning*	*Example*
$\phi(op_1) = op'$, $\phi(op_2) = op'$.	Map distinct source elements onto one destination element.	Translate both `assign` and `initialize` to `store`.
$\phi(op(a, b)) = op'(\phi(a))$.	Drop source operands.	Do not consider type declarations for compilation.
$\phi(op(a, b)) = \phi(a)$.	Prune source operators.	Compile procedure bodies only and forget about headers.
$\phi(op(a, b, c)) = \phi(b)$, *if* $pred(a)$.	Perform static analysis to select operands.	Compile `if-then-else` to its `then` branch, if the boolean condition is always true.

Table 1: Simplifications possible while translating

$$\begin{tikzcd}
Source \arrow[r, "translate"] \arrow[dr, "interpretation"'] & Target \arrow[d, "evaluation"] \\
& Result
\end{tikzcd}$$

Figure 7: Splitting the interpretation

$$\begin{tikzcd}
& \begin{array}{|c|}\hline T_c \\ \hline compile \\ \hline\end{array} \arrow[r, "compile"] & R_c \\
S \arrow[ur, "trans_p"] \arrow[dr, "trans_c"'] & & \\
& \begin{array}{|c|}\hline T_p \\ \hline pretty \\ \hline\end{array} \arrow[r, "pretty"] & R_p
\end{tikzcd}$$

Figure 8: Non-interfering semantics

and possibly layout functions. In a second step the, e.g., string representation of a pretty-print is produced from this structure. Note that now it is perfectly alright to implement the semantics of the target structure as member methods of the target structure. Since that structure is meant for only one purpose, as opposed to the abstract syntax tree which has many interpretations, there is no drawback involved. Figure 8 shows the separation between interpretations and their associated target structures (T_c and T_p). The operation names inside the target structure boxes denote their definition as member methods. There will be no new interpretations on target structures that would require to open and change the whole set of their classes. The interpretation can be defined locally for each element, exploiting inheritance and redefinition, without the disadvantages mentioned in section 2.2.

Besides the nice partition between translation- and semantic related code, the target structure also may serve as a logical structure for the final representation. For instance, a mouse click onto a keyword could cause highlighting the whole corresponding statement and its subcomponents. While this could be achieved by back-pointers into the original abstract syntax tree as well, it is cleaner and more appropriate (as no reinterpretation is necessary) to refer to the pretty-print structure. This argument becomes more obvious in case of interpretations whose logical structure bear less resemblance to the abstract syntax tree (e.g., type-checking information).

Also, assuming multiple users are working simultaneously on one abstract syntax tree, multiple intermediate structures allow them to, e.g., use different compilation options to achieve various code results. If semantic results are stored directly in the source structure, this would be a cause for interference.

Furthermore, an intermediate structure is also helpful when aiming for incremental updates.

- *Incrementality.* Naturally, target structures are subject to fast incremental recomputation since they are produced by homomorphic mappings from their source structure (see section 1). Assuming N to be the number of source structure elements, the asymptotic amount of recalculation needed to update the result of an abstract interpretation is reduced from $O(N)$ down to $O(\log N)$. However, it is necessary to store previously computed results somewhere. Our approach of non-intrusive addition of interpretations forbids storing these at the source elements. The target structure, however, (see figure 6; the flash signs denote places of change) can serve as a store for incremental results. The target structure can play the role of an Observer [Gamma 94] that becomes notified by changes in the source structure and starts the necessary recomputations. The target structure, in turn, is observed by the final result.

- *Space overhead.* If incremental evaluation is not an issue and space efficiency is a priority, then the intermediate structure should be avoided in favor of a direct translation to the final semantics. In a distributed setting, however, we may purposely want to trade space for speed. The intermediate structures can be made local to their users, while the single source structure is accessed only when needed. Ergo, frequent evaluations of target structures do not add to net traffic.

2.8 Implementation

Here is one issue to consider when implementing Translator (also see section 2.10 *Collaboration & Implementation*):

- *Mapping elements to functions.* Translator uses runtime type information (RTTI) to determine the concrete type of a generic function argument. This mechanism is very language dependent and may also vary with different compilers for one language. Usually, it is possible to obtain a string of the class name or test for successful downcasting to a specific type. If no such mechanism is available, one is left with explicitly programming a type inquiry interface. This is, however, incompatible to the otherwise non-intrusive nature of Translator.

  In any case, a generic function may also dispatch on values of objects as opposed to their type. Consequently, you may represent musical notes and quarter notes by the same class. The corresponding objects will differ in a value, e.g., of attribute `duration`. Nevertheless, it is still possible to use a generic function to dispatch on this note representation.

2.9 Sample Code

In the following we present Eiffel code since it is very readable and Eiffel features garbage collection. However, you may use Translator with any language featuring runtime type information such as Smalltalk, C^{++}, and Java.

Assume a toy source language with an if-statement (see figure 3):

```
class    TOY_IFTHEN
inherit  TOY_LANG
creation make
feature
exp, stat:  TOY_LANG;
...
end
```

The corresponding pretty-print element could be:

```
class    PP_IFTHEN
inherit  PP_LANG
creation make
feature
pexp, pstat:  PP_LANG;

make (e, s : PP_LANG) is
  do
    pexp:=e;
    pstat:=s;
  end

display is
  do
    io.putstring ("IF ");
    pexp.display
    io.putstring (" THEN ");
    pstat.display;
    io.putstring (" END")
    io.new_line;
   end;
```

Now we need the specialized function that maps an if-statement to its pretty-print element.

```
class PRETTY_FUNCTION_IFTHEN
inherit FUNCTION[TOY_IFTHEN, PP_IFTHEN]
creation make
feature
genFunc: GEN_FUNC[TOY_LANG, PP_LANG];

make(s: GEN_FUNC[TOY_LANG, PP_LANG]) is
  do
    genFunc:=s
  end;

  infix "@"(ift: TOY_IFTHEN) :
        PP_IFTHEN is
  do
    !PP_IFTHEN!Result.make
      (genFunc @ ift.exp,
       genFunc @ ift.stat)
  end;
end
```

The creation argument of type GEN_FUNC specifies the generic function to be used for evaluation of subcomponents (exp and stat). Its generic[4] parameters denote the function type to be going from TOY_LANG to PP_LANG.

The method for function application (@) simply creates the pretty-print element while supplying the results of recursively evaluating the subcomponents (exp and stat).

The client code for performing a full interpretation is:

```
source:           TOY_LANG;
pp_structure:     PP_LANG;
pretty_functions: PP_FUNCTIONS;
prettyPrint:      GEN_FUNC[TOY_LANG,
                           PP_LANG];
...
!!pretty_functions.init;
!!prettyPrint.make (pretty_functions);

pp_structure:=prettyPrint @ source;
pp_structure.display;
...
```

Prior to its usage, a function package must be initialized by calling init. Then, a generic function (prettyPrint) is created by suppling a pretty-print function package (pretty_functions)[5]. Next, the generic function is applied to the source structure, yielding

[4]This time generic means (static) parametric polymorphism, whereas we imply (dynamic) inclusion polymorphism in case of generic functions.

[5]It is easy to shield the client from the existence of function packages, by providing a tailored generic function that creates a standard generic function with a fixed function package (e.g., pretty_functions).

a target structure (pp_structure). The semantics are finally produced by invoking (display) on the target structure.

A concrete function package appears as follows:

```
class    PP_FUNCTIONS
inherit  FUNCTIONS[TOY_LANG, PP_LANG]
creation init
feature
init is
  local
    pf_var:      PF_VAR;
    pf_assign:   PF_ASSIGN;
    pf_ifthen:   PF_IFTHEN;
    prettyPrint: GEN_FUNC[TOY_LANG,
                          PP_LANG]
  do
    make(3);
    !!prettyPrint.make(Current);
    !!pf_var;
    !!pf_assign.make(prettyPrint);
    !!pf_ifthen.make(prettyPrint);
    put(pf_var,    "TOY_VAR");
    put(pf_assign, "TOY_ASSIGN");
    put(pf_ifthen, "TOY_IFTHEN")
  end;
end
```

Each concrete package inherits from an abstract function package class which, in turn, inherits from HASH_TABLE:

```
deferred
class FUNCTIONS[SOURCE, TARGET]
inherit HASH_TABLE[
          FUNCTION[SOURCE, TARGET],
          STRING]
feature
init is deferred end;
end
```

So, make(3) initializes the function package to a hash table with three entries. Next, a generic function is created in order to serve as the creation argument for the three specialized function prototypes. The function to print variables (pf_var) does not need to recursively evaluate subcomponents, ergo it does not require a generic function for its creation. Note that the Current argument in the creation of the generic function causes the very function package that is currently being initialized to become the argument for the generic function that is supplied to the specialized functions. Finally, the specialized function prototypes are put into the hash table using their corresponding source element class names as keys.

Therefore, the application method of the generic function definition —

```
class    GEN_FUNC[SOURCE, TARGET]
inherit  FUNCTION[SOURCE, TARGET];
         INTERNAL;
creation make
feature
functions: FUNCTIONS [SOURCE, TARGET];

make (fs: like functions) is
  do
    functions:=fs
  end;

infix "@" (source: SOURCE): TARGET is
  do
    Result:=
      clone(functions.item
             (class_name(source))
           ) @ source
  end;
end
```

— can simply access the class name of the source element (method class_name is inherited from the system class INTERNAL), use it to retrieve the correct specialized function prototype (call item on the function package)[6], and then apply a cloned exemplar to its own argument. Instead of a hash table we also could have used a dictionary or even a type case switching statement in order to achieve dispatching on the type of arguments.

[6]At this point a runtime error may occur due to a missing specialized function. Some exception handling or other kinds of gracefully dealing with such a situation would be appropriate.

2.10 Related Patterns

2.10.1 Categorization

- *Interpreter:* Interpreter suggests inventing and representing small languages for reoccurring problems [Gamma 94]. Translator already presumes the existence of an abstract syntax representation. Translator is well-suited to defining the interpretation part of Interpreter which defines interpretations in member methods of elements (see section 2.2) or by using Visitor.

- *Visitor:* Visitor [Gamma 94] has similar motivations as Translator. Yet, besides the fact that Visitor does not cover homomorphic and incremental translations, it also uses a different means of achieving double-dispatch. Visitor relies on the straightforward technique of encoding an arguments type into method names [Ingalls 86]. However, several disadvantages are aligned with this approach:
  - A mutual dependency cycle is introduced between elements and visitors [Martin 97]. This impacts recompilation and regression tests.
  - The elements are forced to know about interpretations because of the need to provide an `Accept` method.
  - One has to provide code for interpreting all elements of a hierarchy, although only a subset will actually be considered by certain interpretations [Martin 97].

- *Facet:* Like Translator this pattern aims at supporting the addition of new and unforeseen interfaces to existing classes without impacting clients that do not require the new interfaces [Gamma 97]. Thus, both patterns preserve the initial key abstraction, i.e., allow element interfaces with intrinsic properties only. Also, both patterns allow for dynamic extensions of classes. Facet differs in that it mainly aims at role modeling and does not take translations or incrementality into account.

- *External polymorphism:* An alternative way (cleverly exploiting C^{++} templates) to achieve polymorphism with classes wich have no common ancestor [Cleeland 96].

- *Acyclic Visitor:* The dependency cycle in the original Visitor design [Gamma 94] caused many suggestions for improvements such as Acyclic Visitor [Martin 97] and Dynamic Visitor [Nordberg 96]. Both alternatives also address the issue of partial visitations, i.e., when a particular interpretation does not need to be defined on all elements. Translator may deal with such a situation as well, since there is no obligation to provide a complete set of specialized functions.

- *Serializer:* Because Serializer [Riehle 97] is a specialization of Visitor, it is also related to Translator. One can think of Serializer as translating objects to a flattened form (e.g., for persistence). In fact, Translator might be more appropriate in some cases since it does not require objects to know about their ability to be serializable.

2.10.2 Collaboration

- *Composite:* Composite structures can be interpreted with Translator [Gamma 94].

- *Observer:* A chain of Observers [Gamma 94] can be employed to account for the data dependency between source structure, target structure, and target semantics.

- *Flyweight:* Abstract syntax tree leaves can be represented as Flyweights [Gamma 94].

2.10.3 Implementation

- *Function Object:* The specialized functions that map source elements to target elements are Function Objects [Kühne 96].

- *Generic Function Object:* The generic functions capable of realizing double- and multi-dispatching interpretations on heterogeneous data structures are generic Function Objects [Kühne 96].

- *Prototype:* Function packages contain specialized function prototypes [Gamma 94]. Further instances are created by cloning and the associated generic function is a preset attribute value.

- *Singleton:* Instead of using an attribute `genFunc` (see section 2.9) in specialized functions, these may alternatively access a Singleton class [Gamma 94] in order to retrieve their corresponding generic function. Eiffel allows for a particularly easy solution by using the "once" mechanism.

3 Conclusion

We presented a general approach to add interpretations to data structures. Especially with regard to the definition of languages, for instance motivated by applications of the Interpreter pattern, our use of first-class functions nicely corresponds to the well-known formal method called denotational semantics. Our functions define local semantic definitions but also — almost invisibly — take care of traversing the source structure.

However, the Translator pattern may be applied in other areas as well. The IRIS Inventor toolkit [Strauss 93] interprets three-dimensional scenes with a double-dispatch scheme relying on run-time type information too. Translator can define interpretations for CAD objects such as displaying, finite element analysis, and cost calculations. More generally, Translator may be used to assign semantics to STEP product data models [Owen 93]. Transferring only the semantics shields a client from the model as a whole. Incremental updates of the semantics makes remote interpretations (e.g., via the Internet) feasible.

We used the generic Function Object pattern in order to provide a mapping from heterogeneous elements to their corresponding translation functions. We, thus, avoided to demand that the source structure know about interpretations. This is valuable for adding semantics to already existing classes that can not or should not be changed. Also, this particular approach of external polymorphism successfully avoids the need to create a dependency cycle between source structure and new interpretation (as is the case with Visitor [Martin 97]).

Furthermore, we demonstrated the utility of homomorphic translations with regard to incremental evaluation. The specific nature of homomorphic mappings naturally allows for incremental updates. We introduced an intermediate structure from which the final semantics are produced. This nicely separates translation and semantics functions, provides a (non-intrusive) place for incremental results, and sometimes is useful for providing a logical structure for computed results.

Although the use of runtime type information is usually not recommended, we have shown it to be of great advantage in this case. Our emulation of multi-dispatch does not involve the disadvantages of the other approaches which we mentioned. Even beyond the capabilities of multimethods in a statically compiled language, our generic functions may register new specialized functions at runtime. The possibility of runtime errors due to non-existent specialized functions is a nuisance, however.

It is recommended that one account for such situations with error handling and check the completeness of function packages manually.

As opposed to object-oriented decomposition, we used first-class functions in order to provide new interpretations. This might appear wrong to object-oriented purists. Note, however, that similar designs (the so-called Command pattern) are suggested and used by others as well [Meyer 88, Gamma 94]. In our case, issues like functional extension (avoiding to change all elements for new interpretations), handling heterogeneous interfaces, iteration of source structures, and non-intrusive addition of interpretations, could all nicely be dealt with, using the single concept of a generic (multi-dispatching) function.

4 Acknowledgments

The author would like to thank Prof. W. Henhapl for the initial spark, Alberto Pardo for support and corrections, Donette Reis for checking the English, and the anonymous referees for their suggestions to improve the paper.

References

[Aho 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, Reading, Massachusetts, March 1986.

[Cleeland 96] Chris Cleeland, Douglas C. Schmidt, and Timothy H. Harrison. External polymorphism — An object structural pattern for transparently extending C++ concrete data types. In *Preliminary Proceedings of PLoP '96*, 1996.

[Gamma 94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture.* Addison-Wesley, 1994.

[Gamma 97] Erich Gamma. The facet pattern. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, 1997. Addison-Wesley.

[Ingalls 86] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Proceedings OOPSLA '86*, pages 347–349, November 1986.

[Kühne 96] Thomas Kühne. Recipes to reuse. In *The 1st Annual European Conference on Pattern Languages of Programming, EuroPLoP '96*, Kloster Irsee, Germany, July 1996.

[Martin 97] Robert C. Martin. Acyclic visitor. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, 1997. Addison-Wesley.

[Meyer 88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[Nielson 93] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* Wiley, 1993.

[Nordberg 96] Martin E. Nordberg. Variations on the visitor pattern. In *Preliminary Proceedings of PLoP '96*, 1996.

[Owen 93] Jon Owen. *STEP: An Introduction.* Information Geometers, 1993.

[Riehle 97] Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. Serializer. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, Reading, Massachusetts, 1997. Addison-Wesley.

[Schmidt 86] D. A. Schmidt. *Denotational Semantics. A Methodology for Language Development.* Allyn and Bacon, Inc, Boston Mass., 1986.

[Strauss 93] Paul S. Strauss. Iris inventor, a 3D graphics toolkit. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pages 192–200. ACM Press, September 1993.

[Wechler 92] Wolfgang Wechler. *Universal Algebra for Computer Scientists.* EATCS 25. Springer-Verlag, Berlin, Heidelberg, New York, 1992.