Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

Denis S. Gurchenkov, Andrey V. Mogilev, Pavel E. Pavlov

ON THE OBJECT-ORIENTED DESIGN OF TRANSLATOR COMPONENTS

Preprint
107

Novosibirsk 2003

This paper discusses the problem of object-oriented compiler construction, with a bias to the design of components operating on abstract syntax tree (AST) and responsible for certain compilation phases. Employing traditional modular design, it would be natural to organize such components as modules. However, in the case of canonical object-oriented design, the code previously put in a separate module has to be distributed over a variety of classes. This may negatively affect further system maintainability. This work describes a novel approach that combines advantages of both modular and OO techniques of the compiler design. The proposed method has proved its viability during the creation of production level compiler components. An implementation technique for the advocated approach using modern object-oriented languages (e.g. Java, C++) is also described.

Keywords: formal semantics, formal model, imperative language, Abstract State Machines, formal methods, storable locations.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Д. С. Гурченков, А. В. Могилев, П. Е. Павлов

ПРИМЕНЕНИЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА К РАЗРАБОТКЕ ТРАНСЛЯТОРОВ

Препринт
107

Новосибирск 2003

В работе рассматривается возможность применения методов объектно-ориентированного проектирования к созданию трансляторов с традиционных императивных языков. Особое внимание уделяется компонентам транслятора, работающим с представлением программы в форме абстрактного синтаксического дерева. Применение традиционного подхода приводит в выражению каждого компонента как отдельного модуля, в то время как ОО-подход диктует разделение каждого компонента на множество классов, распределенных по нескольким единицам компиляции. В работе описывается комбинированный (bi-directional) подход к проектированию компонентов, позволяющий сочетать методы декомпозиции, присущие как традиционному, так и объектно-ориентированному стилю проектирования. Кроме того, обсуждаются проблемы реализации, возникающие при использовании индустриальных языков программирования (C++ и Java).

1. INTRODUCTION

Object-oriented design (OOD) [2] addresses fundamental aspects of software systems creation such as strict coding discipline, ease of modification, maintenance and code reuse, etc. The next step after the modular design [4], OOD is widely acknowledged as a suitable technology for the design of Graphical User Interfaces, class libraries [17], reusable software components [3], and so on. However, the application of OOD to compiler construction has been less investigated so far. On the one hand, using OOD is justified because compilers and translators are large long-living systems whose problem domain entities may be neatly expressed in OO terms. On the other hand, translator systems have a distinctive feature — they include a wide variety of sophisticated algorithms. For instance, modern optimizing compilers [14][6] employ numerous algorithms for static analysis, type inference, register allocation, code selection, etc. Having a multi-component architecture, such systems benefit from traditional modular design, shaping optimization algorithms into separate modules. As a rule, the algorithms share common data, namely the internal program representation. Obviously, that violates encapsulation, a fundamental OOD principle, which requires a tight coupling of code with data. This paper presents a bidirectional decomposition, an approach to the translator system design that combines both OO and modular technique.

The solution is preceded by an overview of translator design techniques currently used in the industry, with emphasis on the on pros/cons of each one.

The rest of the paper is organized as follows: Sections 2 and 3 describe the typical architecture of a translation system and the requirements on the design of its components. Section 4 sets out the known approaches. The proposed design technique is presented in Sections 5 and 6. Sections 7 and 8 describe certain aspects of translator implementation in modern OO languages. Section 9 highlights related works and, finally, Section 10 summarizes the paper.

2. TYPICAL TRANSLATOR ARCHITECTURE

Let a translator be a program that automatically transforms source text written in a programming language to another form. Typical examples are compilers, source-to-source converters, metric compilers, static analysis tools. We consider only translators for imperative programming languages, because the overall translator architecture strongly depends on the family to which the input language belongs.

A typical translator consists of the following components:

- Intermediate Representation (IR)
- Front-end (FE), source program parser and IR builder
- Middle-end (ME), IR analyzer and transformer
- Back-end (BE), output generator
- Project system, or "driver"

Note that the design of the front-end and project system components mostly depends on translator purposes, execution environment, interface of the parser generator used, etc. That is, the design of these components and of the other parts of a translator have only a minor interference. Further, we focus on the elaboration of an intermediate representation and IR-processing components (ME, BE).

We realize that the above description is applicable to simple (one-step) translators, which exploit only one kind of IR (so called single-IR translators). Among them are most of source-to-source converters, source analyzing tools etc. More complex translators (e.g. highly optimizing compilers [6]) are built as chains of basic blocks, where each block is a single-IR translator. In such case, the back-end of each block becomes the front-end of the next one.

We consider the architecture of a single-IR translator that uses an intermediate representation in the form of an AST (abstract syntax tree), also called attributed parse tree [1], [7], [11]. For our purposes it would be sufficient to give an informal definition of AST using OO terminology. AST is based on

- Class inheritance, to structure the IR according to "is-a" relations between the input language entities.
- Object relations (associations), to tackle semantic relations between the entities.
- Encapsulation, to hide low-level implementation details.

3. REQUIREMENTS ON TRANSLATOR DESIGN

We recognize the following specific features of translators:

- Production translators are large systems. Booch et al. [2] state that structural design fails if the amount of source code exceeds approximately 100,000 lines. A typical translation system is of comparable size, so the application of object-oriented design is encouraged.

- Translators used in the software industry live over ten years. They are among the longest-living software products. Therefore, easy maintenance is one of the crucial requirements on such systems.
- During their lifetime, translators incorporate support for several input language dialects and target environments. Ideally, they also have to be updated to employ the most current analysis and optimization techniques. Thus, a well-designed translator should be flexible to a considerable extent.
- Dynamic replacement of components is often desirable, for instance, in multilingual translator systems [13].

In the following sections, we consider approaches to the translator design, taking the above requirements into account.

4. KNOWN APPROACHES

Known approaches to translator design can be split into two categories: algorithm-centric and data-centric. Translation systems that fall into the first category may be considered as a set of algorithms, sequentially applied to data (an intermediate representation). For instance, a typical sequence may be: semantic checking followed by constant evaluation, variable lifetime analysis and, finally, code generation. In data-centric systems, input language entities are represented as classes, and translation algorithms are implemented as class behavior. An example of such a system would be a Java source-to-bytecode translator that contained classes named *Variable, Type, Expression* with methods *semantic_ check(), inline(), generate()* and the like. In this section we discuss the advantages and shortcomings of both approaches in detail.

4.1. Algorithm-Centric Approach

Since the systems exploiting the algorithm-centric approach may be thought of as a "data processing conveyer", they have to address three main issues:

1. Choice of a suitable IR (format of "conveyer data").
2. Component segregation (an exact set of "conveyer components").
3. IR processing (how components operate on IR elements).

IR. So far, the following IR architectures have been established:

- Structural IR, a canonical form representing a program as a graph of 'nodes' where each node is an instance of a structure type. The semantics of particular structure fields often depends on the value of a 'selector' field. Union type instances are frequently used to represent different entities.

- Object-based IR[1]. The characteristic feature of this approach is separating the internals of IR elements from data access interfaces. For instance, IR element properties may be encoded as bit array (represented as an integer field), and accessed through analyzing methods, thus providing data abstraction.
- Object-oriented IR in which objects' classes form an inheritance hierarchy. We would call this approach "pseudo object-oriented" because AST objects have state but no associated behavior.

The implementation language often dictates a particular IR organization, but an object-oriented IR is preferable.

Component Segregation. Another important question is component decomposition, that is, identifying the set of processing components and the tasks they are responsible for. We believe that a simple partitioning into FE/ME/BE is not sufficient to get a flexible system. In fact, a translator may include smaller sub-components, such as:

- Analysis algorithms
- Optimizing transformations
- Semantic checks
- Constant evaluation
- Transforming IR to another form suitable for the next component.

IR Processing. The last topic to discuss in this section is the means of interaction between processing components and IR. The main problem arising here is that components have to dynamically recognize the actual types of IR elements. Any approach to IR design assumes generality of IR elements, achieved through union types, common base types and so on. This implies that processing components have to identify the actual IR elements they deal with. In other words, their implementation always includes some kind of dispatching code, which maps IR object properties to actions of processing components. Typical examples:[2]

[1]Booch et al. identify this as object rather than object-oriented design, since inheritance is not used.

[2]The examples use a mix of Java and C++ syntax, because using pure Java, as we do in other sections, would essentially increase code size.

- Usage of property methods or selector fields of IR objects:
  ```
  switch(node.getKind()) {
      case IntegerType: ... // here follows the handling code
      case PointerType: ...
  }
  ```
- Usage of dynamic type information or the meta-programming technique:
  ```
  if(node instanceof IntegerType)
      ... // handling logic follows here
  else if(node instanceof PointerType)
      ...
  else ...
  ```
- Usage of the “visitor” design pattern [9] with callback methods:
  ```
  interface Visitor { ... }

  // Methods from a base and derived IR classes:
  abstact void Type::accept(Visitor v);
  void PointerType::accept(Visitor c) {
      c.visitPointerType(this);
  }
  void IntegerType::accept(Visitor c) {
      c.visitIntegerType(this);
  }

  // Code of a processing component that
  // initiates the dispatching:
  // (visitor is instance of a class that
  // implements Visitor interface)
  node.accept(visitor);

  // Examples of the visitor methods:
  void VisitorImplementation::visitPointerType(PointerType node) {
      ... /* handling logic follows here */
  }

  void VisitorImplementation::visitIntegerType(IntegerType node) {
      ... /* handling logic follows here */
  }
  ```

Although the application of object-oriented programming style in the algorithm-centric approach is possible, it is rather used as a means of coding and does not affect the architecture of the whole system. Thus we conclude that the algorithm-centric approach adheres to the modular design, as can be seen from the list of benefits and drawbacks.

The benefits are:

- Translation rules and algorithms may be clearly identified. This eases the creation of flexible multi-purpose translation frameworks (e.g. [13]) and results in the higher levels of maintainability and code reuse.
- Dynamic replacement of components can be achieved, including the support for many input languages and target platforms, extensibility with new analysis and optimization techniques, etc. Although the processing components may be implemented as objects (several instances of the same class defining a component interface), we find that it is an example of the modular design.

Among the drawbacks are:

- Violation of the encapsulation principle, because code and data are distributed across different components. This leads to a "fragile" software design — any change in the IR element semantics entails a (possibly unpredictable) number of changes in the processing components.
- Absence of well-defined rules for the intra-component design.

There are a number of compilers and compiler-related tools [8][13][21] built in accordance with this approach [3].

4.2. Data-Centric Approach

According to the OOD principles, IR and processing code should be coupled together. This is the key idea of the data-centric approach. Taking the approach, IR is an abstract syntax tree designed using object-oriented techniques. Front-end builds the program internal representation formed as a system of interrelated objects, whose classes correspond to the input language entities. Inheritance is used to provide generality, e.g. both "binary expression" and "unary expression" classes are derived from a "generic expression" class. Aggregation is employed to represent the hierarchy of entities — variables belong to a function, functions belong to a module, etc. Thus, the translator architecture is a graph of objects, whose relations and type structure are similar to the syntax structure of the input language. Translation algorithms are implemented as methods, e.g. each class has methods like *generate(), const_ evaluate()*, etc. Eventually, we get a translation system as IR with associated behavior.

The advantage of the data-centric approach is a well-designed class and

[3]These references actually present massive compiler systems that consist of several "basic blocks", each of the block can be viewed as application of the algorithm-centric design.

object hierarchy. Processing code may be decomposed and bound to the respective IR objects providing the desirable level of encapsulation. This essentially eases the process of adding new language features and the modification of their semantics. The obvious drawback of the approach is that each translation algorithm is scattered among multiple methods of IR classes, whereas different algorithms are mixed together in the same class. In particular, this hinders reuse of the translator's code. Papers on Aspect-Oriented Programming refer to this drawback as to *code tangling* [22].

We conclude that the data-centric approach is only suitable for relatively simple translation systems that neither require high flexibility, nor incorporate complex translation algorithms, and therefore avoids the mentioned problem.

The data-centric approach is not as widespread as the algorithm-centric approach. One of the main reasons, to our opinion, is that the apporach relies on object-oriented design, and therefore, cannot be applied if the implementation language does not support the OO paradigm of programming.

We have encountered its application oin several research and educational projects [11][12] and the Sun javac compiler [17] which is a good example of a single-purpose translation system.

4.3. Visitors

In the last years, with the development of parser generators targeting OO languages, new standard for interface to internal representation has been established, employing a Visitor design pattern [9].

One can say that the use of the pattern helps the developer to combine benefits of the algorithm-centric and data-centric approaches. Different analysis and translation algorithms, represented as classes that implement the Visitor interface, are separated into different components. At the same time, each component is naturally partitioned into methods that visit concrete classes of IR.

An example: if an internal representation contains instances of classes UnaryOperation and BinaryOperation, then a processing component will include the methods visitUnaryOp and visitBinaryOp.

Our work may be interpreted as an extension of this idea by decomposition of the processing code into classes instead of methods and addition of dynamic relationships between the IR and processing code.

5. BIDIRECTIONAL TRANSLATOR DECOMPOSITION

As can be seen, the sets of benefits and drawbacks of both approaches are orthogonal to each other. This fact is implied by the difference between the system decomposition principles used. Each problem domain entity that is considered as a whole abstraction in one approach is spread among abstractions used in the other. Let us consider the example shown in Fig. 1. In the algorithm-centric approach we may identify the following abstractions: front-end, IR (consisting of Module, Type and Variable classes), semantic checker and code generator. In the data-centric architecture, only one of them is present, namely, the front-end. IR elements become parts of more complex objects, which form the entire translation system. Functionality of the semantic checker and code generator is implemented in the methods of those objects.

Whereas the data-centric approach identifies IR classes (module, type, variable, statement) with associated behavior as abstractions, their counterparts in the algorithm-centric system are distributed over IR, semantic checker and code generator.

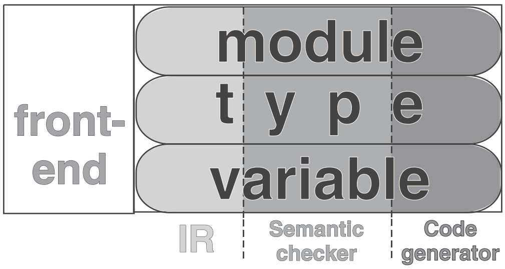

Fig. 1. Decomposition principles

5.1. Decomposition Principles

We propose to combine the approaches and to obtain a system design that amalgamates the advantages of both of them. To this end, we consider a translator system from two points of view at the same time:

- as a set of components (front-end, IR and processing components)
- as a graph of objects whose relations and class hierarchy reflect the syntax structure of the input language

We call the algorithm-centric approach a vertical decomposition, and the data-centric one — a horizontal decomposition. Each may be sequentially applied to the system being designed. We have found that the resulting design (i.e., sets of classes, objects, components, and their communication rules) does not depend on which of the methods is used first.

Vertical-First Decomposition. At first, we divide the system into several components responsible for the main translation stages (or algorithms), exactly as described in Section 4.1. Notice that we separate IR into a distinct component. After that, we design each processing component as a graph of objects, whose relations and class structure are similar to that of IR.

The key observation made here is that the nature of the AST implies further decomposition of each component to blocks, where each block performs the analysis and transformation of one AST element. An example of the resulting design is shown in Fig. 2.

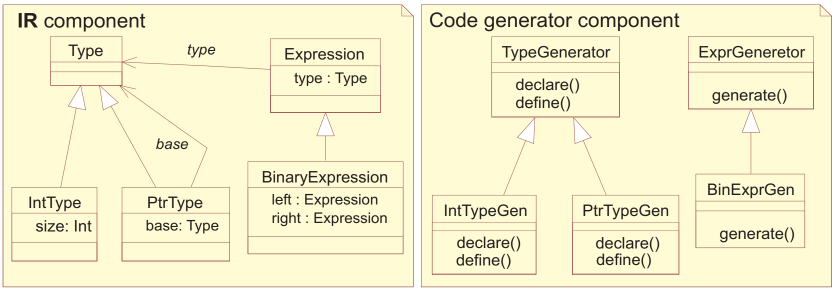

Fig. 2. Example of the system architecture

Horizontal-First Decomposition. Following the data-centric approach, let us consider the translator as a system of AST objects. Then we provide each AST class with methods that implement the necessary transformations of class instances during translation. Note that this implies extension of AST objects with additional states (attributes) that were not present in the original IR, as defined in Section 4.1. The behavior of each class is naturally partitioned into groups of methods so that each group is dedicated to a specific translation phase. We call such group a *facet*. Thus, for each AST class we define a *core facet* that implements the original IR properties, and *operating facets* providing the specific translation functionality. In

Fig. 2, entities *Type*, *PtrType*, *IntType* are core facets, and *TypeGenerator*, *PtrTypeGen*, *IntTypeGen* are operating facets.

The main characteristics of the operating facets are listed below:

- They introduce a new state and behavior to the original IR elements (core facets). Note that the behavior may be very complex, depending on the translation rules employed.
- An operating facet essentially depends on its core counterpart, but not vice versa. That is, the relations of core and operating facets are similar to that of the base and derived classes. In other words, the operating-to-core interface is 'protected' rather than 'public'.
- Relations between different operating facets are considerably weaker.

We have noticed that each operating facet can be designed as a separate class, which is tightly coupled to its core. Then we can group operating facets together into components according to their functionality and, finally, we have the same system design as in the vertical-first decomposition.

6. COMPOSITE OBJECTS AND CLASSES

Following the advocated principles, we decompose a translator into several class hierarchies, each of them corresponding to a specific processing component. Unfortunately, the decomposition violates the encapsulation principle, since we distribute the AST's data and code (state and behavior) among different classes. That leads to a complicated interdependency between components and makes the system difficult to maintain.

This is why we prefer not to think of operating facets as stand-alone classes until we have elaborated the overall design thoroughly. As mentioned above, operating facets are tightly coupled to core ones, so a more fruitful approach would be to exploit the idea of composite objects and classes.

A composite object is a combination of a single core facet and several operating ones. This allows us to minimize superfluous object dependencies. If each facet is an object, a composite object is a tuple consisting of N+1 objects, where one of them represents the core facet and the rest — operating facets. It implies a simple definition of a composite class as a combination of N+1 classes — one for core objects, and N — for operating ones.

As composite classes contain core facets, they directly correspond to the entities of the input language. Thinking in terms of composite classes and objects, one may notice that modification of the input language semantics causes either the introduction of new composite classes, or a change in implementation of existing composite classes. As a rule, the inter-class communication protocol remains unchanged. Similar conclusions are drawn in [5].

On the other hand, modification of a particular translation algorithm (analysis, optimization, etc.) affects only the code of one operating facet for each IR class, leaving other facets almost unchanged.

Summarizing the above, *a translator becomes a system of composite objects.*

Below, we argue that the low-level design of processing components slightly deviates from the "one facet — one object" principle in order to achieve more flexibility in facet relationships. Hence, the composite object is in fact an abstraction used at the design phase, that allows us to make system decomposition simple and effective.

6.1. Proposed System Design

Now we describe the design of a translator in the terms introduced in the previous section. A translator consists of:

- project system,
- front-end,
- IR.

IR is a system of composite objects, whose class hierarchy is organized as an AST. Each composite object consists of N+1 facets, where one is a core facet and the other are operating facets. The number N of operating facets is equal to the number of the processing components identified at the specification and design phases.

As implemented in an imperative OO language (e.g. Java, C++, Oberon-2), each core and operating facet is a separate object. Below we apply the terms "core" and "operating" to both objects and their classes.

Operating classes related to a single processing component form an inheritance hierarchy similar to the hierarchy of core classes. Since composite objects are distributed across components, support for composite objects is implemented through object references. Processing components may be shaped with the aid of the modular facilities of the implementation language (e.g. Java packages, C++ compilation units and namespaces, Oberon-2 modules etc).

Initially, the front-end builds an IR as comprising only core facets. After that, operating facets are created on demand, following the logic of the translation process. The code that creates operating facets and attaches them to core ones is the only translator part that depends on both core and operating facets. We place this code into processing components (along with facet classes) making the core completely independent of the other parts of the system. Thus, the architecture possesses the benefits of the algorithm-centric approach.

The advantages of the proposed system design are:

- Flexible design of class and object relationships w.r.t. multiplicity:
  We are free to link a single operating object with two or more core ones, for instance, if it is stateless, or has a reduced state as in the fly-weight pattern [9]. This is an example of a many-to-one relationship at the object level.
  If two core objects of the same class are to be processed by essentially different rules, we are free to provide them with operating objects of different classes, for instance:
  A core class *BinaryExpression* is used to represent operations on both integer and floating-point values. Let us consider a processing component that performs code generation for Intel x86. Code generation rules for integer and floating operations differ significantly, so using two separate operating classes (*IntegerBinopGenerator* and *FloatBinopGenerator*) would be appropriate. This is an example of a one-to-many relationship at the class level.
- Easy extension of the system with new translation algorithms (in the form of processing components), and with new input language entities (represented as composite classes).
- Dynamic configuration of the system — the required components are created at runtime following the logic of the translation process.
- Opportunity for code reuse due to fine-grained system decomposition.

Note that application of the design to a translator makes its system architecture somewhat complicated. For instance, the design applied to a three-phase translator for a language with a hundred of abstractions (such as C), results in a system containing several hundred classes and many lines of auxiliary code[4], which obviously makes the source text less comprehensible.

We deem that this drawback appears when the translation code is not semantically rich. In this case, the design-level abstractions (represented as classes and objects) are transformed into classes with rudimentary behavior.

As it happens, real-world translators usually incorporate complex algorithms, and the syntactic overhead imposed by our design "dissolves" within the translator code.

[4]Class headers, creation of operating facets, linking them with core, etc.

7. IMPLEMENTATION NOTES

We are aware of dynamically-typed languages (e.g. Self [18], Smalltalk [10]), that would allow us to implement composite objects in a more elegant manner. It is sufficient to note that operating objects dynamically inherit the core ones. However, due to practical considerations, we were restricted to using either Java or C++.

The proposed design was used in implementation of two industrial translators in Java. This section shortly describes the technique of composite objects representation in Java and the problems we encountered during the implementation.

7.1. Referencing composite objects

Since each composite class is a tuple of facet classes, a reference to a composite object is represented as a tuple of references (to facet objects).

An interesting aspect is implementation of the methods of composite classes. In terms of an implementation language, each of the methods belongs to a facet class. Let us consider the hierarchy of three composite classes *Type, IntType, PtrType* as shown in Fig. 2. Suppose that each composite class consists of a core facet and a generator (operating facet), which has an instance method *define()* that emits a type definition clause to the output file (this is an example of a source-to-source converter):

```
CompositeType = <Type, TypeGenerator>
void TypeGen::define(TypeGenerator this, Type core);
```

Note that according to the Java syntax, the 'this' parameter is hidden. We expose it only for a more illustrative description. The method *define()* overridden in the derived classes should have a different signature, because it always receives parameters of the derived facet classes:

```
void IntTypeGenerator::define(IntTypeGenerator this, IntType core);
```

Unfortunately, most of industrial programming languages (e.g. C++, Java) require strict matching of the overridden method signatures, with the only exception being the 'this' parameter. This is why implementation of such methods in the derived classes requires an explicit cast:

```
void IntTypeGenerator::define(Type core0) {
    if(core0 instanceof IntType) {
        IntType core = (IntType)core0;
        // access to the members of 'core'
    } else {
        throw new Error("Composite object integrity broken!");
    }
}
```

We find that type casts hinder writing a compact code. Extension of the implementation language with new constructs may help us to elaborate a more elegant style for programming in terms of composite objects.

7.2. Creation of Composite Objects

We have employed the principles described in Section 6 for decomposition of translators to objects, classes and components. The core and operating objects were associated through references (that is, references to operating objects are attributes of the AST). The front-end builds an AST consisting of “pure” core objects, whereas operating ones are created on demand. We used Java metaprogramming facilities to determine the type of an operating object to be created for a given core class (a “class → class” mapping table is maintained).

After the operating facet is built, its methods can be called. Such calls are performed not directly, but through a special interface designed for the communication of tree-processing components.

An example. The semantic-checking component is working on an expression and wants to find out whether its value can be evaluated at compile time or not. It is aware of the existence of a const-evaluating component, but knows nothing about its internals:

```
package xxx.yyy.semantifier;
class BinaryChecker extends ExpressionChecker {
  public boolean isCorrect(Expression e) {
    BinaryExpression core = (BinaryExpression)e;
    ...
    // need to inquire constancy
    if( Evaluator.facet(core).isConstant() )
        ...
    else
        error("expression must be constant", core.getPosition());
}
```

Here *Evaluator* is the base class for the whole const-evaluators hierarchy and is responsible for maintaining the “core → operating” mapping, which is what the *Evaluator.facet()* method does:

```
public static ExprEvaluator Evaluator.facet(Expression core) {
    ExprEvaluator e = (ExprEvaluator)core.getAttribute(EVAL);
    if(e == null) {
      EvalFactory f = (EvalFactory)table.getFactory(core.getClass());
      e = f.newInstance(core);
      core.setAttribute(EVAL, e);
    }
    return e;
}
```

As can be seen from the example, we have a hashtable whose keys are the classes of AST nodes and whose values are factories responsible for creation of appropriate operating facets. In our implementation, the table is static (filled at startup), but it can be altered dynamically to yield extreme flexibility.

Note that the same result (maintaining the “core → operating” facet relation) can be achieved by static techniques, e.g. double dispatching instead of “class → class” tables, etc.

7.3. Restricted Type Polymorphism

According to Milner’s Type Theory [15], all parameter types are message receivers, i.e. each method is considered as an operation of the types. Then, the method lookup is performed among all of them. The industry standard programming languages implement a restricted version of the concept, for the sake of performance. In particular, only the first parameter type is the message receiver.

We advocate a modified version of polymorphism that counts the types of the method parameters with minimal overhead for execution. The modification affects two language facilities: the method overriding and virtual call. Let us consider a method declared in a derived class that overrides a method of a base class. According to the language syntax, the signature of a derived method must exactly coincide with that of the base method. As shown above, this is inconvenient for implementation of composite objects. In order to overcome the drawback, we propose the following scheme:

- Syntactically, a change of the method declaration rules:
  The signature of the derived method is permitted to differ from the base one in the types of parameters. Now the types may inherit those from the base method signature.

- Semantically, a change of the virtual method lookup:
  when a derived method is invoked, the actual parameters are checked for compatibility with the types of the derived method signature. That should be accomplished for correct execution, because the method signatures may differ. If the types are compatible, the actual parameters are cast and execution proceeds; otherwise a run-time error occurs.

An example.

```
class Type { ... }
class FloatType extends Type { ... }

class TypeGenerator {
    void define(Type parameter);
    }
}

class FloatTypeGenerator extends TypeGenerator {
    void define(FloatType parameter) {
       // use of FloatType properties
    }
}
```

Let the variable *foo* have a formal type *TypeGenerator* and an actual type *FloatTypeGenerator*. Let us consider a method call:

```
foo.define(bar);
```

If *bar*'s actual type is *FloatType*, then execution continues; if not, a run-time error occurs. That may be thought of as "restricted overriding" — a derived class overrides a method declared in the base class, imposing an additional restriction on parameters types. The technique resembles a search for an exception handler matching the type of the exception object.

At first glance, the modification is just a "syntactic sugar" allowing one to write a more compact code. However, changing rules for a virtual method invocation is an extension of the polymorphism concept. Actually, a support for this facility in programming languages would affect designer's "way of thinking" during the development of system architecture. For instance, our design would exploit it for implementation of composite objects.

In our opinion, this kind of dynamic parameter checking may find its place in the next generation of OO languages.

8. PRACTICAL EXPERIENCE

We have built two translators based on the proposed design. This section briefly describes their architecture and lists the exact functionality blocks that are implemented as operating facets.

The translators (named C2F and F2C) form a toolchain translating C source into the F-language, an assembler-like language based on an abstract register machine, and then back to C source.

Each translator incorporates two kinds of IR — the first one (called *primary*) is built by the parser, and is used for program analysis and checking. The second one is hidden inside the backend and is used only for generating some pieces of the output text (such as a C expression).

Each translator incorporates the architecture described above, where the core facets are the nodes of the primary IR, and the operating facets are the classes analyzing and modifying the core.

The primary IR of C2F represents the C program, with class hierarchies for Types, Values, Statements and Expressions, and standalone classes for Variable, Function, Named Constant and Compilation Unit. There are three processing components organized as hierarchies of facet classes: semantic checker, constant evaluator, and back-end (code generator). The semantic checker and back-end contain operating facets for each IR class, while constant evaluator works only with expression and value hierarchies.

There is one processing component that is not organized as a hierarchy of facet classes, this is the alignment and size computation component, called "Aligner". It contains separate pieces of code for each type in the type hierarchy, so we could have implemented it as facet classes, but we decided to use a Visitor pattern [9] instead, because the amount of useful code was too small compared to the associated class headers, "core $\leftrightarrow$ operating" facets relation mechanism, etc. Finally, we had no need for all the flexibility of our approach in this component.

In contrast, using facets for the implementation of back-end and constant evaluator has proven to be the right solution, because the architecture has sustained 2 years of continuous enhancement and still looks well-shaped.

The internal architecture of F2C is quite similar, although it has only two facet hierarchies, for semantic analysis and code generation.

Some operating facets in the code generation component have no state, so they were implemented as flyweights [9], with many core facets referring to the same operating one. Also we've employed the flexibility of object relationships (see Section 6.1) to separate the code generation rules — we have distinct generator objects for the F-language constructions that are instances of the same IR class, but have essentially different representations in the output C text.

An example. The F-language has a notion of "virtual registers" that really are scalar-type variables that have no defined placement and therefore can be accessed only directly by the corresponding register-treatment operations. When translating to C, not all registers are translated to local variables: if a virtual register has only one definition (only one operation writes to this register), and only one use, then the C-equivalent form or defining operation can be substituted to the place of use, therefore the virtual register itself gets no equivalent in the C output.

In practice some code analysis is needed to perform this substitution, but we skip it for brevity.

So, OR of the the F-language has a class "Register" (this is core facet), and the C backend provides an operating facet for this core class. When the code generation logic first requests the generator for a register, the flow analysis is performed that determines how the register should be generated. After that, an operating facet is created and attached to the core one. Depending on the results of analysis, the facet is either an instance of *PermanentRegisterGenerator* or instance of *TempRegisterGenerator*. These classes have common predecessor (an interface), so the rest of the backend treats them identically, while the translation logic encapsulated in the classes is quite different.

Let us finish with a brief source code for this example:

```
class RegisterGenerator {
    public abstract CExpression gen_use();
    public abstract CExpression gen_address();
    public abstract void emit_definition();
    public static RegisterGenerator facet(Register r) {
        // determines what kind of generator is required for
        // the given register and creates it
        ...
        return makeVar ? new PermRegisterGenerator(r) :
                         new TempRegisterGenerator(r);
    }
}

class PermRegisterGenerator extends RegisterGenerator {
    // emits no definition
    // but works with other generators in order to provide
    // C expression for the defining operation of the register
    ...
}

class TempRegisterGenerator extends RegisterGenerator {
    // generates local variable and returns references to that
    ...
}
```

9. RELATED WORK

The idea of the algorithm-centric approach is described in the "compiler construction bible" [1]. Applications of the approach to the production translators may be found in [8][13]. The design of internal representations in the form of ASTs is considered in [7]. A data-centric approach is often employed in educational projects and introductory courses to the object-oriented compiler construction [11][12]. The Sun javac compiler [17] is an example of industrial tool designed using the approach.

From the technical point of view, our work an extension of the "Visitor" design pattern described in [9]. In [20] the pattern is extended by a more sophisticated interface and applied to construction of parser generators. In general, the Visitor pattern expresses each tree-processing component as a class, while we extend it to hierarchies.

As for the translation architecture in the whole, our approach has something in common with the ideas proposed by the authors of the Vanilla framework [5]. They concentrated on the support for many input languages inside the same execution environment (interpreter). Their notion of "pods" as basic blocks of an interpreter is close to what we call "composite objects". However, the key abstractions used in the approaches have different origins — 'pods' are orthogonal (independent) elements of the input language, in contrast to 'composite objects', which are based on an AST. We aimed at combining data-centric and algorithm-centric approaches, while Vanilla is in the vein of the data-centric one.

The translator architecture we propose may be thought of as an example of *multi-dimensional separation of concerns* described by Tarr et al [19]. In their terms, structuring the translator as a hierarchy of IR classes and objects is a "dominant dimension", and each IR-processing component is a "hyperslice" — a unit of decomposition in a dimension other than the dominant.

Multi-dimensional separation of concerns is also a basic principle of Aspect-oriented programming methodology [22][23] which provides a programmer with new abstractions (*Aspects*) used for system decomposition in the dimensions orthogonal to those of classes and objects.

Most of our results can be implemented in the aspect-oriented programming framework, e.g. AspectJ [24][25], possibly, except for dynamics of relationship between the core and operating facets.

However, we stay in the vein of OO programming. New abstractions (facets and composite objects) can be thought of as extensions of classes and objects, or just as another interpretation of the same things (yet with extended relationship between them). In contrast, the aspects are *orthogonal* to the OO features and cannot be implemented through (or expressed with) them [5].

As for implementation, aspects require specific programming environment (source pre-processing is a minimal solution, specific IDE support + compiler + runtime is preferred). As a result, the use of the technology in industrial programming may be inappropriate.

In other words, we've found a point between a widespread, currently used technology (object-oriented) that is ineffective for our needs, and a new programming methodology (aspect-oriented) that is too heavy and not commonly used yet.

[5] One can say that facets *are* objects while aspects *are not*.

10. CONCLUSION

In our experience, the technology described in this paper is particularly useful in two cases:

- Rapid creation of source-to-source converters. Such systems usually exploit only one intermediate program representation in the form of an AST, so the advocated technology is an easy and direct route to the system design.
- Creation of retargetable multilingual optimizing compilers [13]. As noted in Section 2, such compilers are usually built as chains of single-IR translators, or "basic blocks". As a rule, these systems use ASTs as one of their intermediate representations and therefore would benefit from bidirectional decomposition.

In the future we hope to sharpen the technology by applying it to new projects, especially on the following topics:

- Defining more clear rules for the application domain: when a processing component should be decomposed to a hierarchy, and when this is not a good idea.
- Applying our technology to translator components that use other kinds of IR, such as bytecodes, SSA-form [16] etc.

Also we plan to investigate new language constructs allowing better support for programming in terms of composite objects.

Acknowledgments. We would like to thank Vitaly V. Mikheev for his invaluable help in preparing this paper and express our sincere gratitude to Roly Perera for his proofreading assistance. A special thanks to John O. Osbourne for his support.

REFERENCES

1. **Aho A.V., Sethi R., Ullman J.D.** Compilers: Principles, Techniques and Tools. — Addison Wesley, Reading, MA, 1986
2. **Booch G.** Object-oriented analysis and design. — Addison Wesley Longman, Inc, 1994.
3. **Borland** Delphi Development System. — `http://www.borland.com/delphi`
4. **Dahl O.-J., Dijkstra E.W., Hoare C.A.R.** Structured Programming. — NY: Academic Press, 1972.
5. **Dobson S., Nixon P. et al.** Vanilla: an open language framework // Generative and component-based software engineering. — Springer Verlag, 1999.
6. **Mikheev V. et al.** Overview of Excelsior JET, a High Performance Alternative to Java Virtual Machines // Proc. of ACM Workshop on Software and Performance 2002, WOSP'02. — Rome, 2002.

7. **Faigin K. A., Hoeflinger J. P., Padua D. A., Petersen P. M., Weatherford S. A.** The Polaris Internal Representation // Intern. J. of Parallel Programming. — 1994. — Vol. 22, N 5. — P.553–586.
8. **Fraser Ch. W., Hanson D. R.** A Retargetable C Compiler: Design and Implementation. — Addison-Wesley Pub Co., Jan. 1995.
9. **Gamma E., Helm R. et al.** Design Patterns. — Addison Wesley Longman, Inc, 1998
10. **Goldsberg A., Robson D.** Smalltalk-80: The Language and Its Implementation. — Addison-Wesley, 1983.
11. **Holmes J.** Object-oriented compiler construction. — Prentice Hall, 1995.
12. **Justice T. P.** An Object-Oriented Compiler Construction Toolkit. — `ftp.cs.orst.edu/pub/cs-techreports/early/thesis/justice.ps`
13. **Mikheev V. V.** Design of Multilingual Retargetable Compilers: Experience of the XDS Framework Evolution // Proc. of Joint Modular Languages Conference. — Lect. Notes in Comput. Sci. — 2000. — Vol. 1897.
14. **Microsoft** Visual C++ Optimizing Compiler. — `http://msdn.microsoft.com/visualc/default.asp`
15. **R. Milner** A Theory of Type Polymorphism in Programming // J. of Computer and System Sciences. — Elsevier Science. — 1978
16. **Muchnik S. S.** Advanced Compiler Design And Implementation. — Morgan Kaufmann Publishers, 1997.
17. **Sun** JDK 1.3 Documentation. — `http://java.sun.com/j2se/1.3/docs/index.html`
18. **Ungar D., Smith R.** SELF: The Power of Simplicity // Proc. Of OOPSLA'87, Orlando, 1987. — SIGPLAN Notices. — 1987. — Vol. 22, N 12.
19. **Tarr P., Osher H., Harrison W., Sutton S. M. Jr.** N Degrees of Separation: Multi-Dimensional Separation of Concerns. // Proc of the 21st Intern Conf on Software Engineering (ICSE 21). — May 1999, P. 107–119.
20. **Gagnon E. M., Hendren L. J.** SableCC, an Object-Oriented Compiler Framework. // Proc. of Technology of Object-Oriented Languages and Systems Conf., August, 1998. — P. 140.
21. **Aligner G. et al.** An Overview of the SUIF2 Compiler Infrastructure. — 2000. — (Tech. Rep. / Stanford University, 2000)
22. **Kiczales G., Lamping J. et al** Aspect Oriented Programming. // Proc. of ECOOP'97. — Lect. Notes in Comput. Sci. — 1997. — Vol. 1241. — P. 220–242.
23. **Pahlsoon N.** Aspect-Oriented Programming. — `http://www.bluefish.se/aop/aop_niklas_pahlsson.pdf`
24. **AspectJ** programming system. — `http://www.aspectj.org`
25. **Lesiecki N.** Improve modularity with aspect-oriented programming. — `http://www-106.ibm.com/developerworks/java/library/jaspectj/index.html?dwzone=java`

Д. С. Гурченков, А. В. Могилев, П. Е. Павлов

ПРИМЕНЕНИЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА К РАЗРАБОТКЕ ТРАНСЛЯТОРОВ

Препринт
107

Рукопись поступила в редакцию 20.03.2003
Рецензент Г. Г. Степанов
Редактор А. А. Шелухина

Подписано в печать 05.06.2003
Формат бумаги 60×84 1/16 Объем 1,5 уч.-изд.л., 1,7 п.л.
Тираж 50 экз.

НФ ООО ИПО “Эмари” РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6