Electronic Notes in Theoretical Computer Science 82 No. 8 (2003)
URL: `http://www.elsevier.nl/locate/entcs/volume82.html` 29 pages

Some Challenging Typing Issues in Object-Oriented Languages *Extended Abstract*

Kim B. Bruce [1,2]

Department of Computer Science
Williams College
Williamstown, MA 01267, U.S.A.

Abstract

In this paper we discuss some of the remaining problems in the design of static type systems for object-oriented programming languages. We look at typing problems involved in writing a simple interpreter as a good example of a simple problem leading to difficult typing issues. The difficulties encountered seem to arise in situations where a programmer desires to simultaneously refine mutually interdependent classes and object types.

1 Introduction

Early object-oriented languages had weak static type systems (e.g., Simula 67 [1]) or used dynamic rather than static typing (e.g., Smalltalk [14]). The static type systems for more recent languages like C++, Java, and Object Pascal are more rigid than desired, and others like Beta [16] and Eiffel [19] require dynamic or link-time checks to guarantee type safety. Nevertheless, research on the type systems of object-oriented languages (see [3] for a recent survey) has resulted in significant progress in designing static type systems for object-oriented languages that are both safe and expressive. However, there are cases in which standard static type systems for object-oriented languages are not adequate.

In this expository paper we discuss a seemingly straight-forward problem – that of defining a program to process (e.g., evaluate, format, etc.) terms of a simplified programming language. While the language to be processed will be extremely simple, and there are well-known ways of writing such interpreters,

[1] This research is partially supported by NSF grant CCR-9988210.
[2] Email: `kim@cs.williams.edu`

the problem becomes more interesting when we attempt to write the program in such a way that it is easy both to add new expressions to the language and to add new processors that operate on the terms of the language.

The inspiration for the ideas presented here was a discussion on the Java Generics mailing list about the expressiveness of different ways of extending the Java type system. After a discussion by several contributors of possible ways of solving difficult typing problems involving interpreters, Phil Wadler [29] laid out what he referred to as "The Expression Problem" and proposed a solution, which he unfortunately later discovered had a rather subtle error.

The PLT group, originally located at Rice University, but now a rather distributed group, had also investigated several related questions. For example, much of the early part of this paper parallels and in some ways extends the discussion in Krishnamurthi *et al* [15] on ways of combining the strong points of both functional and object-oriented language. However, we point out some additional pitfalls and present some alternative ideas for solutions to the difficulties. As another example, Findler and Flatt [11] discuss similar problems and provide a solution involving the use of units and mixins. We examine more carefully the contributions of these and other PLT group papers later.

We begin in the next section with a brief introduction to the expression problem. In section 3 we investigate how the expression problem would be solved in a functional language like ML. The following two sections investigate two approaches to the expression problem in object-oriented languages. The first is a rather straight-forward approach using what is known as the Interpreter pattern [13], while the second uses the more complex Visitor pattern. In each of these sections we examine limitations of each approach and typing problems that arise when trying to add both new expressions and new processors on the expressions.

In section 6 we sketch a very different approach to solving the typing problems that arose with the Visitor pattern. This approach involves the introduction of syntax to allow the grouping of interface and class definitions so that they may all be extended simultaneously. Following that we discuss some related typing problems whose solutions seems to depend on preserving certain relations between interfaces and classes when extending type groups. We conclude with a comparison with other work in this area and a summary.

2 The expression problem

The problem that we focus on for this paper is that of writing processors on simple programming languages. For most of this paper we will confine ourselves to languages that consist of simple arithmetic expressions.

The first and simplest language consists only of numeric constants and negations. Define:

```
S ∈ SExp ::= n | - S
```

where n stands for a natural number. Thus terms of SExp (for "simple expressions") are either natural numbers or negations of terms of SExp.

We then expand the language to include sums:

```
E ∈ Exp ::= n | - E | E + E
```

We will examine ways of representing terms of these languages as values using both functional and object-oriented programming languages, though our main focus will be on object-oriented languages. We will also examine code to process terms of these expression languages. The two examples of processing terms we will focus on are programs to interpret or evaluate terms of these languages and programs to convert or "format" the terms as strings for display.

All of these are simple to do in a variety of programming languages. What makes our problem more challenging is that we wish to start with a program for interpreting terms of the simple expression language, SExp, and then extend the program both to add new expressions (e.g., by expanding to language Exp), and to add new processors (e.g., the term formatter). Moreover, we would like to perform these extensions by rewriting as little code as possible. This sounds simple, but we will see that it is more difficult than it first appears.

3 Solving the Expression Problem in ML

While this paper is about static typing problems in solving the Expression Problem in object-oriented languages, it is instructive to also look at the solution in a statically-typed functional language, ML. Later we will compare this solution with the most straightforward solution in an object-oriented language.

ML allows users to define new types in a `datatype` statement. Thus we can define a type representing all terms in SExp by writing:

```
datatype sexp = SConst of int | SNeg of sexp;
```

This definition parallels the definition of SExp from above.[3] By convention in ML, type names begin with lower case letters. Thus `sexp` is the name of the new data type. There are two kinds of values in the type, values representing constant terms and values representing negations. The identifiers `SConst` and `SNeg` are constructors used to create the different kinds of values in the type. The `SConst` constructor is applied to a value of type `int` (an integer) to create a value of type `sexp`, while the `SNeg` constructor is applied to a value of type `sexp` to create a new value of type `sexp` that represents the negation of the original. The vertical bar, `|`, is used to separate the descriptions of the two types of values of `sexp`.

A value representing the constant 17 would be written as `SConst 17`. Similarly, the negation of that value would be written as `SNeg (SConst 17)`.

[3] ML does not have a type corresponding to natural numbers, so we must use `int` instead. However we will avoid using negative integers after the `SConst` constructor.

Because of the recursive type definition, we can apply as many `SNeg`'s to a constant as we like. Thus, `SNeg(SNeg(SNeg(SConst 17)))` is also a term of type `sexp`.

With this definition, it is now easy to write a simple interpreter for the terms of SExp as represented by values of `sexp`:

```
fun sinterp (SConst n) = n
  | sinterp (SNeg t) = ~(sinterp t);
```

(The unary minus is written as "~" in ML.) The function `sinterp` has type `sexp` → `int`.

Moreover, it is easy to write new functions to process terms of SExp. For example, we can write a formatter as follows:

```
fun sformatter (SConst n) = Int.toString(n)
  | sformatter (SNeg t) = "-" ^ (sformatter t);
```

Function `sformatter` has type `sexp` → `string`.

Unfortunately, it is not at all easy to expand the datatype to represent all terms of `Exp` and reuse the code of either `interp` or `formatter`. First, the only way to reuse the definition of the definition of `sexp` is to use cut and paste, and even then we must replace the recursive reference to type `sexp` after the constructor `SNeg` with a reference to the new name of the type.[4] Worse luck, if we want to have values of the other datatype remain accessible later in the program, we must also change the names of the constructors:

```
datatype exp = Const of int | Neg of exp | Plus of exp*exp;
```

Thus in the first two cases above, `SConst` became `Const`, `SNeg` became `Neg`, and `sexp` was replaced by `exp`.

Unfortunately, the only way to extend the functions `interp` and `formatter` is to rewrite them from scratch to include the new cases:

```
fun interp (Const n) = n
  | interp (Neg t) = ~(interp t)
  | interp (Plus t u) = (interp t) + (interp u);

fun formatter (Const n) = Int.toString(n)
  | formatter (Neg t) = "-" ^ (formatter t)
  | formatter (Plus t u) = (formatter t) ^ " + " ^
                           (formatter u);
```

Because ML does not allow user-defined overloading,[5] we also had to change the names of these modified functions.

While we could have avoided changing the names of constructors and the function names if we were willing to guarantee that the old and new data

[4] There are experimental extensions of ML that add extensible datatypes, but they have the other problems mentioned.

[5] User-defined overloading would make type inference much more difficult in ML.

types would not be used in the same program, there are occasions where both the new and old definitions both must remain available. In any case we could not simply extend either the datatype definition or function definitions. They needed to be rewritten in full.

The above example illustrates the general phenomenon that it is generally easy to add new functions on a data type in a statically typed functional language like ML, but functional languages typically provide no support for adding new cases to a data type and modifying the functions that were defined on the original data type.

4 Solving the Expression Problem using the Interpreter Pattern

Because Java is likely to be fairly familiar to most readers, we will use Java syntax for writing object-oriented programs. However it should be fairly straightforward to translate our examples to the syntax of other statically type-safe object-oriented languages. Later we will see how extending Java's type system in various ways can increase the expressiveness of the language.

4.1 A simple interpreter in Java

We begin with a straightforward solution to writing an interpreter for SExp by using the Interpreter pattern [13]. The idea is that each kind of arithmetic expression is represented by a different class. However, all of the classes implement the same interface. [6]

The datatype definition from ML is now replaced by an interface definition:

```
public interface Form {
    int interp();        // Interpret formula
}
```

Each of the arithmetic expressions of SExp will be represented by a class that implements `Form`: [7]

```
public class ConstForm implements Form {
    public int value;           // value of constant

    public ConstForm(int value) {
       this.value = value;
    }
```

[6] An interface in Java is like a purely abstract class in C++. However, a class may implement many interfaces, but only extend a single superclass. Hence it is more advantageous to use interfaces than purely abstract classes.

[7] To minimize the size of the classes, we have made the instance variables public, even though that is normally bad style.

```
    public int interp() { return value;  }
}

class NegForm implements Form {
   public Form base;   // formula being negated

   public NegForm(Form basep) {
      base = basep;
   }

   public int interp() { return (- base.interp());  }
}
```

The alternatives in the ML `datatype` definition have now been replaced by separate classes that implement the same interface. A value representing the arithmetic expression "-17" is created and assigned to variable `neg17` by evaluating:

```
Form neg17 = new NegForm(new ConstForm(17))
```

We can evaluate that term by writing `neg17.interp()`.

With this design it is easy to add new kinds of arithmetic expressions. For example we add sums by adding the following class:

```
public class PlusForm implements Form {
    public Form first, second;

    public PlusForm(Form firstp, Form secondp) {
        first = firstp;
        second = secondp;
    }

    public int interp() {
        return first.interp() + second.interp();
    }
}
```

Unfortunately, it is not as easy to add new operations with this design. In ML, we simply added a single new function definition that handled all of the cases. When using the Interpreter pattern, we must extend the interface and all the classes representing arithmetic expressions to include the new method.

Suppose we wish to add a formatting function. As noted above, we must extend the interface and all of the classes implementing it:

```
public interface FForm extends Form {
    String formatter();
}
```

```
public class FConstForm extends ConstForm implements FForm {

    public FConstForm(int value) {
        super(value);
    }

    public String formatter() {
        return "" + value;
    }
}

class FNegForm extends NegForm implements FForm { ... }

public class FPlusForm extends PlusForm implements FForm {
    public FPlusForm(FForm firstp, FForm secondp) {
        super(firstp,secondp);
    }

    public String formatter() {
        return "(" + ((FForm)first).formatter() + " + " +
                     ((FForm)second).formatter() + ")";
    }
}
```

While the extensions to `Form` and `ConstForm` are straightforward – involving just the addition of a new method, the bodies of the method `formatter` in classes `NegForm` and `PlusForm` require the addition of a type cast. We have only illustrated the problem in the newly defined class `FPlusForm`.

The problem is that instance variables `first` and `second` have type `Form`, whereas they need to have type `FForm` in order to be guaranteed that they support the `formatter` method. As a result, a type error would occur if we did not include the casts of `first` and `second` to `FForm` in the body of the method `formatter`. This problem will tend to arise more generally whenever we extend a recursively defined data type.[8]

If the only constructors and methods available were those shown above then the casts in `formatter` will always succeed, because the only place in the code where values are assigned to the instance variables is in the constructor. Because the parameters of the constructor are declared to be of type `FForm`, no values of type `Form`, but not `FForm`, can be assigned to the instance variables. However, if the class `Plus` had included methods to update the left or right summand with a parameter of type `Form`, then the cast in `formatter` might

[8] The recursive nature of the data type is somewhat hidden when written in an object-oriented language, while it is quite apparent in ML. The problem arises with class definitions that mention either the class being defined or one of its interfaces in instance variable or method definitions.

fail at run time.

We would prefer not to have the casts in the method `formatter`. First of all, we would like to guarantee statically that the cast will always succeed. Second, we prefer to avoid paying the unnecessary run-time cost of dynamic checks when we know the cast will succeed.

4.2 Avoiding dynamic casts with LOOJ

We can eliminate the necessity of the type cast by introducing a new `ThisType` (often written as `MyType` in other contexts [3]) type construct. In the language LOOJ [12], the type `ThisType` represents the public interface of `this` inside a class. Like the self-referential `this`, the meaning of `ThisType` changes when a method involving the type is inherited in a subclass. We illustrate the use of `ThisType` by rewriting the `PlusForm` and `FPlusForm` classes so that the instance variables have type `ThisType`.

```
public class PlusForm implements Form {
    public ThisType first, second;

    public PlusForm(ThisType firstp, ThisType secondp) {
        first = firstp;
        second = secondp;
    }

    public int interp() {
        return first.interp() + second.interp();
    }
}

public class FPlusForm extends PlusForm implements FForm {
    public FPlusForm(ThisType firstp, ThisType secondp) {
        super(firstp,secondp);
    }

    public String formatter() {
        return "(" + first.formatter() + " + " +
                     second.formatter() + ")";
    }
}
```

The types of the instance variables `first` and `second`, as well as the parameters `firstp` and `secondp` of the constructor for `PlusForm` have been modified to have type `ThisType`. Inside the class `ThisType` will be interpreted to have type `Form`. Moreover, the constructor may be called with parameters that are actually values of type `Form`.[9] However, the constructor for `FPlusForm` may only be invoked with parameters of type `FForm` because the meaning of `ThisType` shifts in the extension.

Classes that mention `ThisType` are type checked under the relatively weak assumption that `ThisType` is an (undetermined) extension of the public interface of the class being defined. Thus, in `PlusForm`, `ThisType` is treated as an extension of `Form`. As a result, when type checking method `interp`, the type checker knows that `first` and `second` provide the `interp` methods. While we may assign a value of type `ThisType` to a variable of type `Form`, it is illegal to assign a value of type `Form` to the variable `first`, because we may only assume that `ThisType` is an extension of `Form`.

Because `ThisType` is assumed to be an extension of `FForm` when used inside class `FPlusForm`, the invocation `first.formatter()` in the body of method `formatter` is well-typed.

Although we do not have the space to go into details here, the discussion above implies that if there were a method `replaceFirst` in class `PlusForm` that updated the value of `first` then its parameter would have to be of type `ThisType` rather than `Form`. As a result, when inherited in `FPlusForm`, no difficulties would result.

While the above example only used `ThisType` as an instance variable, it can also occur in method parameter and return types. For example, a class can declare a `clone` method that returns a value of type `ThisType`. Similarly, a `compare` method might take a parameter of type `ThisType`.

Message sends of methods that have parameters of type `ThisType` must be handled slightly differently from other methods. To be type-safe, we must know the exact type of the receiver of the message. Thus languages like LOOM and LOOJ include ways of indicating when we know the exact type of an object, rather than its type up to extension. In LOOJ, we indicate this by adding an "@" symbol before the type. Thus an object generated from class `FNegForm` could be assigned to a variable with type `@FForm`, but not a variable of type `@Form`. See [3,6,5] for detailed discussion of type-safety issues with similar languages, and [12] for a more complete discussion of LOOJ.

4.3 Evaluating the use of the Interpreter Pattern in solving the Expression Problem

The Java and LOOJ solutions presented above are based on the Interpreter Pattern [13]. As shown above, it was easy to add new expressions to the formal language to be processed with a program based on the Interpreter Pattern. In going from a program to interpret expressions of SExp to one interpreting expressions of Exp, we only had to add a new class for the sum expressions.

[9] The parameters are given type `ThisType` so that they may be assigned to the instance variables, which also have type `ThisType`.

It was more difficult to add a new operation on expressions of the language. However, we were able to use inheritance to ease the pain to some extent. We were required to extend the `Form` interface and each of the classes implementing it to include the new method (in the case above, `formatter`). While the Java solution required type casts when adding new methods, the LOOJ solution could be statically type checked.

Finally, we note that with this style of solution, it is difficult to restrict a variable to only hold values from SExp, for example. It would be possible to introduce different interfaces for terms representing values of SExp and Exp. For example, classes `ConstForm` and `NegForm` could be defined to implement interface `SForm`, while these two classes as well as `PlusForm` could be defined to implement interface `Form`.

The simplest way of accomplishing this would be to define `SForm` to extend `Form`, as then `ConstForm` and `NegForm` need only be declared to implement `SForm`. While this would be simple to do (just define `SForm` to extend `Form` with no added methods), it would be hard to accomplish if `Form` and `PlusForm` were defined some time after the definition of `SForm`, `ConstForm`, and `NegForm`. The difficulty is that `SForm` would have already been defined as having method `interp`. If the programmer went back and redefined `SForm` to extend `Form` (rather than directly including the method `interp`), all of the classes that mention `SForm` would have to be recompiled. This is certainly doable if all of the source code is still around, but requires more modification of existing code than is desirable.

After the next section, we investigate another approach to solving the Expression Problem by using the Visitor pattern. We will see that using the Visitor pattern makes it easier to add new processors on the arithmetic expression language, but unfortunately, makes it harder to add new kinds of arithmetic expressions.

4.4 Comparing functional and object-oriented approaches

From the above discussions it is clear that functional languages make it easy to add new operations on a data type, but generally make it harder to add new variants. On the other hand, the Interpreter pattern in object-oriented languages makes it easy to add new variants, but relatively hard to add new operations.

John Reynolds [26] was one of the first to compare functional and data-centered approaches to problems. William Cook [9], however, was the first to explicitly compare ADT style approaches (like that of ML) with that of object-oriented languages. Several papers from the PLT group [11,15] have also discussed issues revolving around these differences.

Ideally we would like to find a language or design pattern that would make it easy to add both variants or operations. As a first step, in the next section we examine the Visitor Pattern for object-oriented languages. We will see that using the Visitor pattern makes it easier to add new processors on the arithmetic expression language, but unfortunately, makes it harder to add new kinds of arithmetic expressions. Nevertheless this will provide a different way of looking at the problem that will prove useful when we examine type groups.

5 Solving the Expression Problem using the Visitor Pattern

The Visitor Pattern [13,23] represents a way of defining data structures that have multiple "visitors" operating on them. An important reason for using this pattern is that it makes it easy to add new operations on the data structures.

5.1 Using the Visitor Pattern

In applying the Visitor pattern to the Expression problem, we will include a method `process` with all classes representing arithmetic expressions. The method `process` will take a parameter that represents an object with the capability of performing operations on the arithmetic expressions. Thus if `aexp` represents an arithmetic expression and `interpObj` represents an object capable of interpreting arithmetic expressions, then `aexp.process(interpObj)` would return the value of the expression represented by `aexp`.

A first attempt at an interface for expressions based on using the Visitor pattern might look like:

```
public interface Form {
    // Process formula with visitor lp.
    Result process(BasicLangProc lp);
}
```

Unfortunately, we run into an immediate problem – we need to figure out what the return type, `Result`, of `process` should be. If our language processor is an interpreter, then clearly `process` should return type `int`. However, if it is a formatter, then it should return type `String`. This is usually solved in languages like Java by setting the return type of `process` to be type `Object`, as both integers and strings can be converted to type object (via the `Integer` wrapper case for integers). When an expression is processed with a visitor then the result will need to be converted to the appropriate type via a type cast.

Because we are interested in static type safety, we wish to use a more accurate type system that will allow us to avoid type casts, at least as much as possible. We can do this by adding F-bounded polymorphism to Java as is done in GJ [2] and LOOJ [12]. These language support the use of bounded type parameters in classes, interfaces and methods.

The following provide examples of the use of type parameters in interfaces and methods:

```
public interface Form {
    // Process formula with visitor lp.
    public <Result> Result process(BasicLangProc<Result> lp);
}

// BasicLangProc: Visitor of consts & negations
public interface BasicLangProc<Result> {
    // process constant expression
    public Result constCase(ConstForm cf);

    // process negation expression
    public Result negCase(NegForm nf);
}
```

The interface `BasicLangProc` takes a type parameter `Result` that indicates the type of the answer that will be returned by the language processor. For example, an interpreter will return a result of type `Integer`.[10] Thus the class representing an interpreter will implement `BasicLangProc<Integer>`. A formatter will return a result of type `String`, so the class representing the formatter will implement `BasicLangProc<String>`.

Interface `Form` contains method `process`, which also has a type parameter `Result`. Note that the type parameter for a class is written before the return type of the method. Thus, method `process` takes a type parameter `Result` and a regular parameter `lp` with type `BasicLangProc<Result>`, returning a value of type `Result`.

It will be easiest to explain how the Visitor pattern works by introducing two classes that implement the `Form` interface:

```
public class ConstForm implements Form {
    public int value;    // value of constant

    public ConstForm(int val) { value = val;}

    public <Result>Result process(BasicLangProc<Result> lp) {
        return lp.constCase(this); }
}

public class NegForm<Result> implements Form<Result> {
    public Form pos;    // formula to be negated

    public NegForm<Result>(Form ppos) { pos = ppos; }

    public <Result>Result process(BasicLangProc<Result> lp) {
```

[10] GJ does not support instantiating type variables with primitive types like `int` so we must use the wrapper class `Integer`.

```
          return lp.negCase(this); }
  }
```

In both of these classes, the constructor simply initializes an instance variable. In each case the `process` method is very simple. It simply sends a message to the processor parameter, including itself (`this`) as the actual parameter. The `ConstForm` class sends the message `constCase`, while the `NegForm` class sends the message `negCase`. Thus the actual message sent to the visitor (language processor) corresponds to the kind of formula being interpreted.

Because the interface `BasicLangProc` has a method with a name and parameter type corresponding to each class of arithmetic expression that implements `Form`, the message sends are all type correct. A language processor implementing `BasicLangProc` thus provides methods to handle each kind of argument that it needs to process.

The following two classes represent language processors:

```
  class BasicInterp implements BasicLangProc<Integer>{
      Integer constCase(ConstForm cf) {
          return new Integer(cf.value); }

      Integer negCase(NegForm nf) {
          return new Integer(
              -(nf.pos.<Integer>process(this)).intValue());}
  }

  class BasicFormatter implements BasicLangProc<String> {
      String constCase(ConstForm cf) {
          return Integer.toString(cf.value);

      String negCase(NegForm nf) {
          return "-" + nf.pos.<String>process(this); }
  }
```

As we discussed earlier, the class `BasicInterp` implements the interface `BasicLangProc<Integer>` because it returns values of type `Integer`. The method `constCase` in the method has full access to all of the public features of `ConstForm` (including the public instance variable `cf`) because the parameter to `constCase` is declared to have type `ConstForm`.

The method `negCase` is slightly more complex because it involves an indirectly recursive call to process the instance variable `pos` of `nf`. Because we do not know whether the value of `nf.pos` comes from class `ConstForm` or `NegForm`, we cannot predict in advance which `process` method will actually be executed.

The following code generates terms of type Form representing the constant 17 and the expression -17, as well as an interpreter, `binterp`, and formatter, `bformatter`. It also shows how each can be used.

```
Form cf = new ConstForm(17);
Form nf = new NegForm(cf);

BasicLangProc<Integer> binterp = new BasicInterp();

... nf.<Integer>process(binterp) ...;

BasicFormatter<String> bformatter = new BasicFormatter();

... nf.<String>process(bformatter) ...;
```

For readers not familiar with the use of the Visitor pattern, we provide a trace of what happens when the `process` method is sent to `nf` with `binterp` as parameter.

```
nf.<Integer>process(binterp)
    calls
binterp.negCase(this) where this is nf
    returns
new Integer(- nf.pos.<Integer>process(this).intVal())
      where nf.pos is cf and this is binterp and
       cf.<Integer>process(binterp) returns new Integer(17)
    returns
new Integer(-17)
```

We can see from this trace that flow of control bounces back and forth between the arithmetic expressions and the language processor. Essentially the Visitor pattern uses a form of "double dispatch" in order to ensure that the correct code is chosen based on both static (in the invocation of `constCase` and `negCase`) and dynamic (in the invocation of `process`) dispatch.

By design, adding new operations (processors) is easy with this design. What about adding new expressions? Adding a new sum expression involves not only adding a new class to represent the new kind of expression, but also requires extending the `LangProc` interface and all existing processor classes:

```
interface LangProc<Result>
                  extends BasicLangProc<Result>{
    Result plusCase(PlusForm pf);
}

public class PlusForm implements Form {
    Form first, second;  // sum parts

    PlusForm(Form fstp, Form sndp) {
        first = fstp; second = sndp;
    }
```

```
        <Result>Result process(BasicLangProc<Result> lp) {
            return ((LangProc<Result>)lp.plusCase(this);
        }
    }

    public class Interp extends BasicInterp
                        implements LangProc<Integer> {

        // return sum of two pieces.
        public Integer plusCase(PlusForm pf) {
            int firstVal =
                (pf.first.<Integer>process(this)).intValue();
            int secondVal =
                (pf.second.<Integer>process(this)).intValue();

            return new Integer(firstVal + secondVal);
        }
    }

    public class Formatter extends BasicFormatter
                           implements LangProc<Integer> {
        // return formatted sum.
        public Integer plusCase(PlusForm pf) {
            return pf.first.<String>process(this) + " + " +
                   pf.second.<String>process(this);
    }
```

Extending `LangProc`, `BasicInterp`, and `BasicFormatter` are all relatively straightforward. Each simply needs a new method to handle objects of class `PlusForm`.

Somewhat surprisingly, it is the `PlusCase` class itself that requires a type cast for safety. This time the problem is not a matter of updating the type of an instance variable, as it was for adding operations using the Interpreter pattern. Instead the problem is that the type of the parameter to the `process` method does not reflect the added capabilities of interface `LangProc`. Instead, the parameter's type only guarantees that it supports the `constCase` and `negCase` methods, while the method body requires that it support the `plusCase` method.

What is needed here is to be able to specialize the parameter type to `LangProc` rather than using `BasicLangProc`. Unfortunately this is not allowed in Java, as it could lead to the failure of type safety.

In fact, as pointed out in Krishnamurthi *et al* [15] with a different example, things get even worse if a visitor needs to generate a new visitor on recursive calls. This would be required, for example, if our interpreter was extended to handle function expressions. In that case the interpreter would need to maintain an environment, and the interpretation of function applications would generally require the creation of new interpreter objects with updated environments to provide values for formal parameters when evaluating function bodies. To handle this either a special `This` constructor (see [12]) or a carefully designed Factory class would be required to create new interpreters inside the methods of both BasicInterpreter and Interpreter. See [15] for a further discussion of these problems.

5.2 Can the Visitor pattern be made statically type safe?

As usual we would like to find a way to avoid this type cast so that we can be guaranteed static type safety. However, we are not allowed to change the types of parameters to a method. Moreover, even extending the language with a `ThisType` construct does not help as the type of the parameter is not directly related to the class or its interface. The only way that we can change the type is to make it a type parameter.

Unfortunately, even making the type of the parameter a type variable does not solve the problem as the parameter of method `process` is itself parameterized by `Result`. Thus rather than using a type variable, we will need to use a variable that represents a function from types to types.

Moreover, the cause of the typing problem is that different classes implementing `Form` have different requirements on the kinds of processors that they need. Thus different kinds of arithmetic expressions will require different kinds of language processors. Thus as a first step, we provide the following new interface:

```
public interface Form<Visitor extends BasicLangProc> {
    // Process formula with visitor lp.
    <Result> Result process(Visitor<Result> lp);
}
```

The parameter `Visitor` to `Form` represents a function from types to types.

Now the classes `ConstForm` and `NegForm` can use any processor that has at least methods `constCase` and `negCase`. On the other hand, the class `PlusForm` requires processors that also provide the method `plusCase`. Thus we define:

```
public class ConstForm<Visitor extends BasicLangProc>
                                implements Form<Visitor> {
   public Integer val; ...
   public <Result> Result process(Visitor<Result> lp) {
       return lp.constCase(this); }
}

public class NegForm<Visitor extends BasicLangProc>
```

```
                              implements Form<Visitor> {
   public Form<Visitor> base; ...
   public <Result> Result process(Visitor<Result> lp) {
       return lp.negCase(this); }
}

public class PlusForm<Visitor extends LangProc>
                               implements Form<Visitor> {
    public Form<Visitor> left, right; ...
    public <Result> Result process(Visitor<Result> lp) {
        return lp.plusCase(this); }
}
```

To provide some intuition as to why these classes should be parameterized by the type function of the processor, note that if `nf` is an object from class `NegForm<BasicLangProc>` then the instance variable `base` must have interface `Form<BasicLangProc>`. In particular, `base` may not hold a term that requires anything beyond what is contained in `BasicLangProc`. Thus `base` may certainly not hold a value generated by `PlusForm`.

On the other hand, if `nf` is an object from class `NegForm<LangProc>` then the instance variable `base` has type `Form<LangProc>`, and hence may hold a value from `PlusForm<LangProc>`, for example. Thus the type function parameter to these classes specifies the complexity of the processor required to interpret the arithmetic expression – including all of its subexpressions.

So far so good. But now it gets worse. How do we write the actual visitors? Here is a first attempt at the interface `BasicLangProc`.

```
interface BasicLangProc<Result> {
    // process constant expression
    Result constCase(ConstForm<???> cf);

    // process negation expression
    Result negCase(NegForm<???> nf);
}
```

Recall that classes `ConstForm` and `NegForm` are parameterized by a visitor interface. Looking at the series of call-backs when the code is executed, it is clear that the methods need classes that can deal with the same processor as is being defined. That is, the question marks in the interface definition could be replaced by `BasicLangProc`. That would work fine for writing interpreters and formatters for `ConstForm` and `NegForm`, but we would also like to extend `BasicLangProc` to `LangProc` that adds a `plusCase` method. Inside `LangProc` each of the methods needs to take parameters that are formulas that can be handled by `LangProc` rather than `BasicLangProc`, so we need to change the types of parameters – something that is illegal.

We seem to be caught in a bind here. There are several options that might lead to success. One is to introduce a `ThisTypeFcn` construct that, when used in a parameterized class, would represent the parameterized interface of that class. Then `ThisTypeFcn` could replace the question marks in the `BasicLangProc` interface:

```
public interface BasicLangProc<Result> {
    // process constant expression
    Result constCase(ConstForm<ThisTypeFcn> cf);

    // process negation expression
    Result negCase(NegForm<ThisTypeFcn> nf);
}
```

As with `ThisType`, the meaning of `ThisTypeFcn` would change in extensions, providing the flexibility desired above.

Another alternative that may be helpful is to add more type variables and support contravariant changes in parameter types (including the bounds in bounded polymorphism). However, both of these alternatives require fairly exotic type theories and are arguably too complex for programmers to easily understand and use.

Solutions are possible that have interface `Form` parameterized as well by the type of the result of the visitor, but that is not a satisfactory solution because then different representations would be necessary to handle visitors with different result types. For example an object of type `Form<Integer>` could handle an interpreter visitor, but not a formatting visitor.

At this time I don't know how to write a type safe solution to the Expression problem using the Visitor pattern that avoids type casts and that uses well understood type systems (including type parameters and `ThisType`, but not including higher-order type variables or higher order `ThisType`). In the next section we examine a completely different approach that seems to lead to a more understandable solution.

6 Mutually interdependent object types and classes

One of the problems with the use of the Visitor pattern in the previous section is that when a new variant of the data type (e.g., a new arithmetic expression) is added, we must also extend all of the existing processors of the data type. The typing rules make it difficult to make these additions sequentially without generating static typing errors. Instead, we would prefer to have them happen simultaneously, and to have changes to the one show up automatically in references to the other.

In an earlier paper [8], we introduced the notion of type groups in LOOM. The type groups allowed us to group together mutually recursive types in such a way that all could be changed simultaneously Here we sketch how this same idea could be added to LOOJ, and hence to Java. We concentrate on displaying the ideas with a new solution to the Expression problem. Technical details will be given elsewhere.

The following is a sketch of the code for a type group named `Basic`. It incorporates an interpreter and formatter for the the basic language SExp. Recall that `@T` is the type representing values whose type is exactly `T` and not an extension. The exact types are included in the code below to ensure type safety.

```
group Basic {
    public interface Form {
        <Result> Result process(@LangProc<Result> lp);
    }

    public interface LangProc<Result> {
        Result constCase(@ConstForm cf);
        Result negCase(@NegForm nf);
    }

    public class ConstForm implements @Form {
       public Integer val; ...
       public <Result> Result process(@LangProc<Result> lp) {
          return lp.constCase(this); }
    }

    public class NegForm implements @Form {
        public @Form pos;    // value of constant
        ...
        public <Result> Result process(@LangProc<Result> lp) {
            return lp.negCase(this); }
    }

    public class Interp implements @LangProc<Integer> {
        public Integer constCase(@ConstForm cf) {
            return new Integer(cf.val);
        }

        public Integer negCase(@NegForm nf) {
            return new Integer(nf.pos.<Integer>process(this));
        }
    }

    public class Formatter implements @LangProc<String> {
        public String constCase(@ConstForm cf) {
            return Integer.toString(cf.val);
        }
```

```
            public String negCase(@NegForm nf) {
                return - + nf.pos.<String>process(this);
            }
        }
    }
```

There are no surprises in the above code, as the only difference from the original version using the Visitor pattern given in section 5.1 is the addition of exact types.

Adding a case for sum expressions is now easy:

```
    group Full extends Basic {
        public interface LangProc<Result> { // extends old
            Result plusCase(@PlusForm pf);
        }

        public class PlusForm implements @Form {
            public @Form first, second;
            ...
            public <Result> Result process(@LangProc<Result> lp) {
                return lp.plusCase(this);
            }
        }

        public class Interp implements @LangProc<Integer>{
            public Integer plusCase(@PlusForm pf) {
                return new Integer(
                    pf.first.<Integer>process(this)+
                    pf.second.<Integer>process(this));
            }
        }

        public class Formatter implements @LangProc<String> {
            public String plusCase(@PlusForm pf) {
                return pf.first.<String>process(this) + " + " +
                       pf.second.<String>process(this);
            }
        }
    }
```

Notice that we have reused the interface name `LangProc` and class names `Interp` and `Formatter`. Because group `Full` extends group `Basic`, interfaces and classes of `Full` automatically extend the interfaces and classes of `Basic` that have the same name.[11] Thus interface `LangProc<Result>` contains the `constCase` and `negCase` methods from the corresponding interface in `Basic`, as well as the newly declared `plusCase` method.

When new methods are added to an interface in an extending group, all classes implementing that interface in the original group must be extended in the extending group in order to satisfy the type checker. Thus, classes `Interp` and `Formatter` must be extended in group `Full` to include the `plusCase` method added in `LangProc<Result>`.

Alternatively, classes can be omitted from the groups and defined separately. In that case we require that the group name be added to the interface being implemented. For example, we could write

```
    public class BInterp implements Basic.@LangProc<Integer> {
public Integer constCase(@ConstForm cf) {
    return new Integer(cf.val);
}

public Integer negCase(@NegForm nf) {
    return new Integer(nf.pos.<Integer>process(this));
}
    }

    public class FInterp extends BInterp
                        implements Full.@LangProc<Integer>{
public Integer plusCase(@PlusForm pf) {
    return new Integer(
pf.first.<Integer>process(this)+
pf.second.<Integer>process(this));
}
    }
```

This is possible because it is only the interfaces that are mutually referential. The classes need only refer to the mutually referential interfaces.

The choice of whether or not to include classes in the group will depend on the intended use. If new operations are to be added later, for example, it would be most convenient to add those outside of the group rather than extending the group. If that is not likely, then it is notationally a bit simpler to include the classes within a group. Either style can be supported.

Type-checking of a class `C` in the group is done assuming only that the interfaces and class names from the group mentioned in `C` are (possibly) extensions of the definitions appearing in the group. As a result, we can prove that no type problems can arise with inherited code when a group is extended.

It is straightforward to use the constructs defined in groups. The following

[11] If desired, it would be easy enough to require the extension to be declared explicitly.

declares some formulas and operations and then invokes the operations on the formulas.

```
@Basic.Form bconst = new Basic.ConstForm(17);
@Basic.Form bneg = new Basic.NegForm(bconst);

@Basic.Formatter bformatter = new Basic.Formatter();

... bneg.<String>process(bformatter)) ...;

@Basic.Interp binterp = new Basic.Interp();

... bneg.<Integer>process(binterp) ...;

@Full.Form fcon1 = new Full.ConstForm(17);
@Full.Form fcon2 = new Full.ConstForm(13);
@Full.Form fplus = new Full.PlusForm(fcon1,fcon2);

@Full.Interp finterp = new Full.Interp();

 ... plusf.<Integer>process(finterp) ...;
```

Notice that the type of a formula indicates its group. This is important because that group determines the strength of the language processor that must be used with it. For example, a negation of a constant can be operated on by a term of type `Basic.LangProc<Result>`, whereas a negation of a sum, can only be operated on by a term of type `Full.LangProc<Result>`.

The only unfortunate part of this solution is that we had to seed the code with occurrences of "`@`" in order to assure type safety. It would have been preferable to be able to say that a constant term could be handled by any operation that had at least the capability of `Basic.LangProc<Result>`. However, there are similar examples where removing these exact types leads to a violation of type safety. In general it is important not to mix and match components from different type groups as type insecurities pop up quite frequently if that is allowed. The removal of such annotations in specialized cases is part of our ongoing research.

Type groups seems to be quite helpful in solving this and other problems involving mutually referential collections of interfaces and classes. Our earlier paper [8] provided a similar example using the Subject-Observer pattern. We remark, however, that the type system presented in that paper was much weaker than it should have been. More recent work has resulted in a system that allows us to type check more complex examples using type groups.

There is an important advantage of this approach, as opposed to approaches using some form of bounded polymorphism. That is that when we use classes and interfaces from the same group, we are guaranteed that we are working with components that match exactly. With bounded polymorphism we get matches up to extension, and subtle problems arise in those cases that can break what seem to be quite reasonable solutions (see [29,28].

7 Further interesting problems

There are a number of other related problems that are currently not easily typable with common static type systems for object-oriented languages. The paper [18] describing the Jiazzi system for adding components to Java suggests some interesting problems that also involve simultaneous extension of interfaces and classes.

One particularly interesting example involves a system of GUI components defined as follows:

```
group GUIComponents {
    interface Component { ... }
    interface Button extends Component {...}
    interface Window extends Component {
        void addComponent(Component item) {...}
    }
}
```

Suppose we extend the group to add a `setColor` method to `Component`:

```
group ColorComponents extends GUIComponents {
    interface Component {
        void setColor(Color newColor);
    }
}
```

We mentioned earlier that any classes that implemented `Component` in the group `GUIComponents` would have to be extended to include the method `setColor`. What about the interfaces `Button` and `Window`? It would be convenient to assume either that they automatically gained the method `setColor` or that the programmer was required to add them to pass the type checker.

That is, we would like to design groups so that they not only preserve uses of other types and the "`implements`" relation, but also preserve the "`extends`" relation. If that were the case then the following code would be type safe:

```
ColorComponents.Window clrWindow;
ColorComponents.Button clrButton;

clrWindow.addComponent(clrButton);
```

If `ColorComponents.Window` did not extend `ColorComponents.Component` then the method call in the last line above would be illegal.

If we insisted that interface extension be preserved in group extensions then we could write bounded polymorphic classes and methods similar to the following:

```
<CompGP extends GUIComponents> void doGUIStuff
     (@CompGp.Button button, @CompGp.Window wind) {
         ... wind.addComponent(button)... }
```

If not, this sort of polymorphism would not add much expressiveness to the language.

An interesting question is what relations should be preserved under group extension.

8 Comparison with other work

As mentioned in the introduction, members of the PLT group originating at Rice University have written several papers examining typing issues related to those discussed in this paper – particularly working with systems of interdependent types and classes. For example, Krishnamurthi *et al* [15] also compare the use of the Interpreter pattern with approaches based on the Visitor pattern. They highlight the contrasting difficulties of adding tools with the first approach and of adding new variants in the second. They present a solution using the Visitor pattern that is similar to that presented in section 5.1, and hence also requires the type cast. They then focus on resolving problems with generating new visitors for recursive calls to the `process` method.

Findler and Flatt [11] also discuss the difficulties of extensibility, and propose a solution based on units and mixins in MzScheme. McDermid *et al* [18] describe a preprocessor for Java that allows a programmer to define units and mixins, and then compile them down to Java. Because they are similar, we only discuss Jiazzi here. In Jiazzi one can define parameterized packages, where the parameters represent packages to be linked in later. These parameterized packages can be used as "mixins", essentially allowing classes to be defined whose supertype is a parameter.

Because these units contain multiple interface and class definitions, they can be used to define and extend collections of mutually referential classes and interfaces. When the programmer wishes to use them, these packages can be explicitly combined using a link statement that states precisely how the pieces fit together. A link statement may involve cyclic links and may involve the explicit formation of fixed points. These link statements can get quite complex, and may prove to be quite difficult to use and understand in practice. Essentially this is like the difference between working with the generators of recursive functions and then specifying fixed points of these generators as opposed to just simply writing down recursive definitions. Making life even more complex, "upside-down" mixins seem to be required to solve problems like those discussed in section 7. Type checking may be performed modularly for each unit, with special requirements to determine if link statements are legal.

Smaragdakis and Batory [27] independently proposed similar ideas supporting what they term as “collaborations” by using templates in C++ to generate “mixin layers”. Because C++ templates are not type-checked until they are instantiated, type checking mixins is not an issue.

A potential advantage of using mixins is that they may be instantiated in different contexts. The corresponding disadvantage is that using mixins for solving problems such as those discussed here is complicated because the programmer must specify the desired packages as explicit fixed points formed from cyclicly composing parameterized packages.

Palsberg and Jay [23] also discuss the Visitor pattern, but focus on making the Visitor pattern more flexible by using reflection to have the visitor itself discover the recursive structure of the object being visited.

Type groups are based on a generalization of the type rules associated with the “`MyType`” construct that is contained in our earlier languages PolyTOIL [7] and LOOM [5] (see [3] for a history of the use of `MyType` constructs in other languages). The construct `MyType` represents the type of `self` or `this`, and, like those terms, automatically changes its meaning in subclasses, thus providing a significant increase in the expressiveness of a static type system.

Type groups were introduced in [8] as a generalization of `MyType` obtained by introducing a collection of mutually interdependent, but distinct “`MyType`”s. When a type group is extended, all of these `MyType`s change simultaneously. Just as there exist a set of provably safe static typing rules for languages that have a `MyType` construct, there also exist a set of provably safe static typing rules for these collections of mutually interdependent `MyType`s.

Type groups add additional expressiveness to the provision of parametric polymorphism (see also [4]). While they capture some of the expressiveness of the virtual classes [17] of Beta, they differ from Beta by providing static type safety, while virtual classes require dynamic checks to guarantee safety.

Type groups are most similar to Ernst’s independently developed Family Polymorphism [10], but are based on a simpler and better understood type theory. In particular, the underlying theory of Family Polymorphism is based on dependent types, which is powerful enough to introduce difficulties with decidable type checking. The language Scala [21] appears to have similar expressiveness, but its type system is provably undecidable.

Eiffel [19] allows covariant changes to the types of instance variables as well as method parameters and return types. this allows programmers to make the kind of changes to types that we have shown are necessary to add new cases and operations using the Visitor pattern. Unfortunately, unrestrained use of these features leads to programs that are not type safe. Chapter 17 of Meyer [20] describes three possible solutions to this problem, each of which would restrict the language in some way. Unfortunately, as far as we know none have been implemented to see what limits arise in the expressiveness of the resulting languages. The approach using groups described in this paper could likely be adapted to make Eiffel statically type safe.

Palsberg and Schwartzbach [22] also discuss the possibility of extending mutually recursive classes. In their framework, all references to fixed classes in a group are assumed to be updated when any in the group is changed. That is, all classes referenced are treated as generalized `MyType`s. They propose a system for checking such programs, but it depends on a “closed world” assumption. That is, one may not modularly check the classes in a program. All code for the complete program must be available to perform the check.

Remy and Vouillon [25] show that the language OCAML [24] can handle the kinds of problems illustrated by the Subject-Observer pattern and other examples similar to the Expression problem. The use of OCAML has the advantage that the different types can be refined separately and then combined. Much of the advantage is as a result of a combination of the use of row types (which allow expression of the equivalent of `ThisType`) and type inference. A disadvantage of their approach is that the resulting types are essentially unreadable, being expressed with bounded quantification where the signature of each type involves type variables from all types involved in the system. As such it can be difficult for a programmer to easily understand how the components can be combined.

Nevertheless their solution does allow separate refinement of each of the components of a system, with assembly into a group of interacting types only taking place later. Our proposal, on the other hand, takes advantage of the fact that, in many of these circumstances, the types are refined as a group. This greatly reduces the complexity of the types as presented to the programmer.

9 Current status and summary

Our current implementation of the language LOOM [5] supports both `MyType` and type groups. The more recent LOOJ [12] extension of Java supports `ThisType`, but has not yet been extended to implement the groups as outlined here. We are currently investigating both practical and theoretical issues involved in having groups preserve a broader range of relations among interfaces and classes in the group.

In summary, type groups provide an expressive, yet statically type safe way of increasing the expressiveness of object-oriented programming languages, especially with regard to programs that involve the simultaneous refinement of collections of mutually interdependent classes and object types. They seem to be useful in combining polymorphism with design patters that involve mutually referential types. We are currently exploring ways of ensuring that a variety of relations between interfaces and classes can be preserved during simultaneous refinement of groups of mutually referential interfaces and classes.

References

[1] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Aurbach, 1973.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM.

[3] Kim Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[4] Kim Bruce, Marin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP '98*, pages 523–549. LNCS 1445, Springer-Verlag, 1998.

[5] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[6] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995.

[7] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *TOPLAS*, 25(2):225–290, 2003.

[8] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Electronic notes in Theoretical Computer Science*, volume 20, 1999. `URL: http://www.elsevier.nl/locate/entcs/volume20.html`, 26 pages.

[9] W. Cook. Object-oriented programming versus abstract data types. In *Proc. of the REX Workshop/School on the Foundations of Object-Oriented Languages (FOOL)*, pages 151–178. Springer-Verlag, LNCS 173, 1990.

[10] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[11] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 94–104, 1999.

[12] John N. Foster. *Rupiah: Towards an expressive static type system for Java*. Williams College Senior Honors Thesis, 2001.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstraction and reuse of object-oriented designs*. Addison-Wesley, 1994.

[14] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation.* Addison Wesley, 1983.

[15] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *ECOOP '98*, pages 91–113. LNCS 1445, Springer-Verlag, 1998.

[16] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The Beta programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. M.I.T. Press, Cambridge, MA, 1987.

[17] Ole Lehrmann Madsen and Birger Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA '89 Proceedings*, pages 397–406, 1989.

[18] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. of OOPSLA*, October 2001.

[19] B. Meyer. *Eiffel: the language.* Prentice-Hall, 1992.

[20] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition, 1997.

[21] Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Informal proceedings of FOOL 10*, 2003.

[22] J. Palsberg and M. Schwartzback. Type substitution for object-oriented programming. In *OOPSLA-ECOOP '90 Proceedings*, pages 151–160. ACM SIGPLAN Notices,25(10), October 1990.

[23] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.

[24] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object-Oriented Systems*, 4:27–50, 1998.

[25] Didier Rémy and Jérôme Vouillon. The reality of virtual types for free! Unpublished note avaliable electronically, October 1998.

[26] J.C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168, 1975.

[27] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[28] Philip Wadler. The expression problem: A retraction. Message to Java-genericity electronic mail list, Feb 11, 1999.

[29] Philip Wadler. The expression problem. Message to Java-genericity electronic mail list, November 12, 1998.