# Visitor-Oriented Programming

Thomas VanDrunen
Purdue University
Department of Computer Science
West Lafayette, IN 47907
vandrutj@cs.purdue.edu

Jens Palsberg
UCLA Computer Science Department
4531K Boelter Hall
Los Angeles, CA 90095
palsberg@ucla.edu

## ABSTRACT

Multiple dispatching and the visitor pattern are approaches to making object-oriented programs more extensible. Both have a flavor of pattern matching, thereby moving object-oriented programming closer to functional programming. The key idea of these approaches can be crystallized as a notion of visitor which lies between functions and objects. Can this idea be developed into a new form of visitor-oriented programming which combines the best of functional and object-oriented programming? As a first step, we present a visitor calculus in which each value is a visitor and every visitor call uses double dispatching. We illustrate the relationships to other paradigms by translating the lambda-calculus to the visitor calculus, and the visitor calculus to Java. Our calculus forms the core of a language in which we have programmed the translation of the lambda-calculus to the visitor calculus itself. To demonstrate the expressiveness of visitors, we show the translation in four versions that use smaller and smaller subsets of the language. Along the way, we present correctness proofs and examples of running an implementation of our language.

## 1. INTRODUCTION

### 1.1 Background

A program is a set of datatypes and a set of behaviors over cases of those datatypes. Maintaining a program requires extending the datatypes and their cases and extending the behaviors over them. We would like to to do this modularly, that is, without recompiling the entire system each time an extension is made. This represents a well-known way of comparing programming paradigms and a dilemma in programming language design [12, 3, 7, 15]. In object-oriented languages, we can easily extend the datatypes by writing new classes, including subclasses. Extending behavior, however, is difficult since methods are members of a class and so cannot be changed or added to without recompiling that class. Functional programming takes the other side of the trade off. There, new functions on a datatype can be written and compiled separately, but once a datatype is created, its cases are fixed.

The visitor pattern [5] is an attempt to solve this problem in an object-oriented paradigm; that is, it is a way to add functionality modularly without changing the set of datatypes. If we have a set of classes and determine that we need new functionality for them, instead of adding a method to each and recompiling, we write a visitor class, with a visit method for each class, which will perform that functionality. This way we add behavior modularly. However, while introducing the benefit of functional programming to an object-oriented setting, we are also importing the liability. Now we can no longer add classes easily. To use visitors for extension, we must in advance give each class an accept method (not to mention making sure that the visitor will have access to all internal data necessary for its task). More seriously, a visitor must know all the classes on which it will operate [11]. The visit and accept methods are necessary because using visitors is essentially using dynamic double dispatch— the correct method is selected based on both the acceptor's class and the visitor's class. Most object-oriented languages support only single dispatch.

What we notice is that while a visitor is in fact an object, in a sense it can also be viewed as a function. A visitor is related to both object-oriented and functional programming. We see a spectrum:

**Spectrum:** function — visitor — object.

Is there a way we can exploit this approach to system building so that we have the advantages of both paradigms, but the disadvantages of neither? What we would like is a language designed specifically for programming with visitors, where the visitor is the basic unit and the need for accept methods and precognition of acceptor classes is subsumed by the language semantics, including rules for double dispatch. In other words, we would like a language where both datatypes and behaviors can be added modularly.

Various systems have been developed to study double dispatch and the visitor pattern. Millstein and Leavens [8] designed the Tuple language to show how to add multiple dispatch to conventional object-oriented languages cleanly, letting the receiver of a method call be a tuple of objects rather than a single object and selecting a method most specific to those objects. If there is no appropriate method or no most specific method, such a call causes a *message not understood* or *message ambiguous* error, respectively. The second of these errors is a result of the dispatch being symmetric. Generalizing to multimethods (any methods where dynamic dispatch depends on more than one argument), a multimethod is *symmetric* if all dispatching arguments are treated uniformly, that is, no argument has priority for determining an appropriate method.

Millstein and Chambers [9] presented Dubious, an object-oriented core language with symmetric multimethods. Dubious uses a module system to reason about extensibility and features a static type system which ensures that programs that type check do not throw *message not understood* or *message ambiguous* errors. Functions are objects and

thus are first-class. Similarly, Clifton, Leavens, Chambers, and Millstein [2] presented MultiJava, also with symmetric multimethods, but with a class system like that of Java. A feature of MultiJava is the open class, a class to which methods can be added without changing the class directly. This offers an alternative to the visitor pattern in that new functionality can be added to a class without the the prior planning needed to make infrastructure for visitors.

In order to capture the flavor of the visitor pattern itself, double dispatching should be *asymmetric*— if a visitor does not have a method that matches the class of an acceptor exactly but inherits a method that does, that method is chosen over a method it has which matches an ancestor class to the acceptor. Asymmetric semantics eliminates the possibility of a *message ambiguous* error since the precedence among arguments disambiguates the method selection. Furthermore, a `visitor` class serving as an ancestor to all visitors could contain a visit method for whatever class tops the class hierarchy (such as `Object` in Java; in a language designed specifically for visitors, this may be the `visitor` class itself) and ensure no message will not be understood. which is convenient for a core calculus and formal reasoning since eliminating the possibility of the computation causing an error allows us to focus on the computation itself. ¿From a practical software engineering point of view, symmetric double dispatching may be preferable; however, in this paper, we will pursue the asymmetric variant, to keep it close to the visitor pattern.

Dubious is classless but uses a module system that is good for what it formalizes—modules; a core calculus that focuses on visitor behavior, however, would be more tractable without them and without the full programming language features of MultiJava. We would further like a calculus that can be related to both functional and object-oriented paradigms by encodings among representative languages, demonstrated by correctness proofs and implementations.

Castagna, Ghelli, and Longo [1] studied the $\lambda\&$-calculus of overloaded functions with multiple dispatching. While motivated by object-oriented programming, the $\lambda\&$-calculus is purely functional. In the $\lambda\&$-calculus, an overloaded function is essentially a set of functions, and an overloaded-function call uses dispatching via the types of its members. Notice that functions are the primitive concept from which overloaded functions are constructed. While the $\lambda$-calculus is a sublanguage of the $\lambda\&$-calculus, translating the $\lambda\&$-calculus to a language such as Java remains unexplored and seemingly difficult. As far as we know, the $\lambda\&$-calculus has not been used as the core of a language.

## 1.2 Our results

This paper presents a new calculus, VisiCalc[1] and a larger language, Peripaton, for which VisiCalc is a core. Both hard-wire and formalize the visitor pattern. These languages will encourage the use of the visitor pattern as a foundational programming paradigm. They also provide a framework for formal reasoning about the visitor pattern, modular extensibility, dynamic double dispatch, and the relationship between object-oriented programming and functional programming. We present translations of the $\lambda$-calculus to the visitor calculus, and of the visitor calculus to Java. Along the way, we present correctness theorems and examples of running an implementation of Peripaton.

### 1.2.1 A visitor calculus

In our visitor calculus, each value is a visitor and every visitor call uses double dispatching. The grammar for VisiCalc is as follows:

| | | | |
|---|---|---|---|
| $p$ | ::= | $(\overline{c}, e)$ | *programs* |
| $e$ | ::= | | *expressions* |
| | | $(e\ e)$ | *invocations* |
| | \| | `new` $t[\overline{e}]$ | *creations* |
| | \| | $x$ | *field references* |
| | \| | `acceptor` | *parameter references* |
| $c$ | ::= | $t : t\{\overline{x};\ \overline{m}\}$ | *classes* |
| $m$ | ::= | $(t \mapsto e)$ | *method mappings* |

A formal semantics will be presented in Section 3. For now, a VisiCalc program is a set of classes and a main expression. Each class defines a visitor type; a class definition contains a name for the class, the class's parent, a list of fields, and a list of methods. Methods do not have names, but are identified by the type of a dispatching parameter (we can think of each method having the name `visit`, and there can be at most one such method for any parameter type). The parameter is referred to by `acceptor`; intuitively, the callee or receiver of the method has a visitor, and the parameter is the acceptor. (One could easily reverse this and call the methods "accept" and the parameter "visitor," if the reader finds that more in line with his experience with the visitor pattern.)

When a method is called, $(e_1\ e_2)$, the method mappings for in the definition of $e_1$'s class are searched for a method matching $e_2$'s type. If none are found, the methods of $e_1$'s parent class are searched, and so forth. If no appropriate method is found among $e_1$'s class and all its ancestors, then we do a similar search for a method appropriate for $e_2$'s parent class. This entire process will terminate because we assume a pre-defined class `visitor` that is at the top of the class hierarchy, as `Object` in Java, which has a method appropriate for any parameter, returning that parameter.

Here is the translation of the $\lambda$-term $\lambda x.x$:

```
cla0: visitor {
  (visitor ->  acceptor )
}

 new cla0[]
```

This and other code examples appear as they are produced by our implementation of the translation algorithms which we give formally in Section 2.1. It has one class, representing the single lambda, which has a method that simply returns its parameter. The main expression is an instantiation of the class. To apply this, we use the main expression as the visitor in an invocation; whatever the acceptor is, it will surely be an instance of `visitor`, and so the method in `cla0` is called, returning the acceptor.

Here is the translation of the visitor program into Java:

[1]Our VisiCalc is unrelated, but named in tribute, to VisiCalc, the first commercial spreadsheet program, published by Software Arts.

```
class visitor extends Object {
  visitor() {}
  visitor accept(visitor x) {
    return x.visit_visitor(this);
  }
  visitor visit_visitor(visitor acceptor) {
    return acceptor;
  }
  visitor visit_cla0(cla0 acceptor) {
    return this.visit_visitor(acceptor);
  }
}
class cla0 extends visitor {
  cla0 (){
    super();
  }
  visitor accept(visitor x) {
    return x.visit_cla0(this);
  }
  visitor visit_visitor(visitor acceptor) {
    return acceptor;
  }
}

new cla0()
```

In object-oriented programming, visitors typically have a visit method and data structures have an accept method. In VisiCalc, everything is a visitor and everything can be visited. Therefore, the equivalent classes in Java have both visit methods and an accept method. All classes in the translated system extend the class `visitor`. The object `new cla0()` is a visitor, and to visit an object (say, another `new cla0()`), we call `(new cla0()).accept(new cla0())`. The accept method calls `visit_cla0()`; `cla0` has no such method in its definition, but it inherits one from `visitor` which simply calls the visit method appropriate for the acceptor's parent class, `visitor`. Finally `cla0.visit_visitor()` is called, returning its parameter, the original acceptor `new cla()`.

### 1.2.2 A language based on the calculus

To show how VisiCalc can serve as the core of a larger language, we extend it to allow extra, non-dispatching arguments to parameters, self references, local variables (defined in let expressions), non-local field references, unique identifiers, and branch expressions. The grammar for the language, called Peripaton, is shown in Figure 1.

Note that $x$ is now called a variable reference rather than a field reference. This is because in Peripaton variables can also refer to local variables and extra parameters. For the sake of Peripaton, we assume unique identifiers are values along with visitors. They are produced by `gensym`. The comparison in branch statements is defined only for unique identifiers; the comparison is true if both values originate from the same evaluation of `gensym`.

### 1.2.3 Discussion of the calculus and language

Several design decisions deserve clarification before we proceed. First, as mentioned earlier, asymmetric semantics may not be desirable—always having a step that can be taken is not necessarily good, if that step is something unintended by the programmer. However, a type system which helps identify bugs is not our purpose here, but only to demonstrate that a middle ground between functional and object-oriented programming exists. Second, the language and the calculus are statically untyped and therefore no casts are needed. While unusual in an object-oriented setting, it is not without precedence, as both the Abadi-Cardelli calculus and the language SmallTalk are not statically typed. Third, although Peripaton allows self-reference in the usual form of a `this` pointer, this construct is strikingly absent from the calculus. All three of these decisions have the same effect: they keep the language and especially the calculus as small as possible, and they anchor VisiCalc closer to the lambda calculus, rather than leaving it adrift as another object-calculus simply having visitor-like syntax. Different design decisions might be expected for a fully-featured language, but that is not our present goal. There are, of course, other language features, such as pattern-matching in ML [14] and Scala [13], which achieve some of the same goals, but do not so clearly resemble a middle-ground between functional and object-oriented programming.

| | | | |
|---|---|---|---|
| $p$ | ::= | $\overline{c}\ e$ | *programs* |
| $e$ | ::= | | *expressions* |
| | | $(e\ e)$ | *invocations* |
| | \| | $(e\ e\ <\overline{e}>)$ | *invocations with extra, non-dispatching parameters* |
| | \| | `new` $t[\overline{e}]$ | *creations* |
| | \| | $x$ | *variable references* |
| | \| | `acceptor` | *dispatching parameter* |
| | \| | `this` | *self references* |
| | \| | `let` $(\overline{x} = \overline{e})$ `in` $e$ `end` | *let expressions* |
| | \| | $\{e\}.x$ | *non-local field references* |
| | \| | `gensym` | *unique identifier creations* |
| | \| | `if` $e = e$ `then` $e$ `else` $e$ | *branch expressions* |
| $c$ | ::= | $t : t\{\overline{x};\ \overline{m}\}$ | *classes* |
| $m$ | ::= | | *method mappings* |
| | | $(t \mapsto e)$ | *ordinary mappings* |
| | \| | $(t <\overline{e}> \mapsto e)$ | *mappings with extra, non-dispatching parameters* |

**Figure 1: The Peripaton grammar**

Our language and calculus support single-inheritance hierarchies. However, for the translation of lambda-terms to VisiCalc, the full power of inheritance and dynamic dispatch is not needed. Since the use of a type hierarchy is well known, our paper does not illustrate it further.

We will later present a translation from VisiCalc to Featherweight Java [6] which, linked with our encoding of the $\lambda$-calculus, provides a translation from the $\lambda$-calculus to Featherweight Java. Our translation does not use inner classes or continuation-passing style. It would have been straightforward to map each function to a class with an "apply" method containing the function's body, but that would require nested classes to represent nested functions. If nested classes are not allowed, a known solution to the problem of translating a higher-order language to a first-order language is to use continuation-passing style transformation and closure conversion [4]. Our translation does use a form of closure conversion; a further investigation of the relationship between our approach and others is left to future work.

**The rest of the paper** Section 2 gives an extended example, a compiler written in Peripaton which translates from the lambda calculus to VisiCalc. This illustrates and motivates the programming paradigm, demonstrates the connection between functional programming and visitor-oriented programming, and leads the reader from Peripaton to VisiCalc. Section 3 gives a formal specification of VisiCalc, including a type system and a theorem of type soundness. Section 4 builds the bridge to conventional object-oriented programming by giving a translation from VisiCalc to Featherweight Java and presenting theorems of type preservation and behavior preservation.

## 2. FROM FUNCTIONS TO VISITORS

To illustrate and motivate the programming paradigm, we present a succinct but realistic example. A familiar use of the visitor pattern is performing compiler passes on an intermediate representation of a program; for example, Gamma et al. [5] use this in their introduction to the pattern. Consider abstract syntax trees; many operations desirable in the middle of a compiler can be implemented by a traversal of the tree (static checking, code transformation, pretty-printing, code generation, etc.). When visiting a node in the tree, the visitor can recursively visit the components (children) of that node and use those results in computing the result at the current node. A new compiler pass can be written quickly by extending a general depth-first visitor, inheriting code to perform the traversal— this is particularly convenient for compiler passes that need to inspect only certain parts of the tree. This is also convenient for maintaining the compiler if new constructs are added to the language: a new compiler can be constructed by extending all the visitors with visitors that contain methods only for the new constructs.

In this section we present a translation from a call-by-value $\lambda$-calculus to VisiCalc. We first specify the translation formally and then implement it in our language Peripaton. To demonstrate the expressiveness of visitors, we present four versions of the implementation, written in smaller and smaller subsets of Peripaton. The four programs are each 200–300 lines long; in this section we will show excerpts. In the full version of the paper, which is available from our webpage, there is an Appendix with all four programs in their entirety, including comments.

### 2.1 Specification of the translation

The metavariable $\Delta$ stands for the "current" variable or, if there is none, is empty. $\Gamma$ is a list of VisiCalc classes. Even though $\Gamma$ contains the whole class, if $\alpha$ is the name of a class in $\Gamma$, we say $\alpha \in \Gamma$ for convenience. $\Delta \cdot \Gamma \vdash M \rightsquigarrow \Gamma' \ e$ means that, given environment variables $\Delta$ and $\Gamma$, lambda calculus expression $M$ is translated to the list of VisiCalc classes $\Gamma'$ and main VisiCalc expression $e$. $fv(M)$ returns the free variables of lambda calculus expression $M$. A variable is *free* in an expression if it is not bound as the argument in any lambda abstraction that occurs in the expression and encloses the variable. If a term has no free variables, then it is *closed*. The notation $[x/y]fv(M)$ means the list of free variables in $M$ with $x$ substituted for $y$.

$$(x) \cdot \Gamma \vdash x \rightsquigarrow \Gamma \ \texttt{acceptor} \tag{1}$$

$$(y) \cdot \Gamma \vdash x \rightsquigarrow \Gamma x, \quad y \neq x \tag{2}$$

$$\frac{\Delta \cdot \Gamma \vdash M_1 \rightsquigarrow \Gamma' \ e_1 \quad \Delta \cdot \Gamma' \vdash M_2 \rightsquigarrow \Gamma'' \ e_2}{\Delta \cdot \Gamma \vdash (M_1 \ M_2) \rightsquigarrow \Gamma''(e_1 \ e_2)} \tag{3}$$

$$\frac{(x) \cdot \Gamma \vdash M \rightsquigarrow \Gamma' \ e \qquad \alpha \notin \Gamma'}{\begin{array}{rcl}(y) \cdot \Gamma & \vdash & \lambda x.M \rightsquigarrow \\ & & \Gamma' \cdot \alpha : \texttt{visitor}\{fv(\lambda x.M) \\ & & \qquad (\texttt{visitor} \mapsto e)\} \\ & & \texttt{new} \ \alpha[[\texttt{acceptor}/y]fv(\lambda x.M)]\end{array}} \tag{4}$$

$$\frac{(x) \cdot \Gamma \vdash M \rightsquigarrow \Gamma' \ e \quad \alpha \notin \Gamma' \quad fv(\lambda x.M) = \emptyset}{\begin{array}{rcl}\emptyset \cdot \Gamma & \vdash & \lambda x.M \rightsquigarrow \\ & & \Gamma' \cdot \alpha : \texttt{visitor}\{(\texttt{visitor} \mapsto e)\} \ \ \texttt{new} \ \alpha[]\end{array}} \tag{5}$$

This algorithm translates a program by building up a list of classes and producing a main expression. Each lambda term represents a new class in the list of classes and an expression creating an instance of that class. Intuitively, functions become visitors. The free variables in the lambda term become fields in the class, and the argument to the abstraction is the argument to a method in the class (i.e., `acceptor`). For this reason, we need to know the argument to the nearest enclosing lambda term, if any, represented by the metavariable $\Delta$.

Rule (1) translates an occurrence of the current variable to `acceptor`. Rule (2) leaves an occurrence of a non-current variable intact; it becomes a field reference. To translate an application, rule (3) translates the two composing lambda calculus expressions (each of which translation may have the side effect of producing some new classes, captured by the use of $\Gamma'$ and $\Gamma''$) and creates an invocation composed of the the two resulting visitor expressions; thus function calls are translated to function calls.

The translation of an abstraction when a current variable is present in rule (4), results in a new class being added to the class list, as we anticipated above. Note that the class hierarchy is flat, with all classes extending `visitor`. The single method dispatches on `visitor`, meaning that this method will apply to any argument, as would be the case for the original lambda term. The body of that method is the translation of the body of the lambda term, and that translation may itself produce new classes. The arguments given to the constructor are the free variables of the term (note that this excludes $x$) since the presence of a current variable implies that this will be a subexpression of a body of a method and so fields will be present. We need to replace the current variable with `acceptor`, however, since the method parameter, rather than a field, is associated with it. Intuitively, the expression `new` $\alpha[\dots]$ builds a closure that represents $\lambda x.M$. Rule (5) covers the case for a lambda term where there is no current variable and the free variable list is empty. (If there is no current variable, then we must be translating a top-level term, and if the program we are translating is a closed term, then the free variable list will be empty. The translation is not defined for a top-level term that is not closed.) In that case the constructor needs no arguments since the class has no fields.

The second example is a term for application, $\lambda f.\lambda x.fx$. This takes two arguments (in curried form) and treats the first as a function, applying the second to it. It is translated to

```
cla0: visitor {
  f;
  (visitor -> ( f acceptor ))
}
cla1: visitor {
  (visitor ->  new cla0[acceptor])
}

 new cla1[]
```

Each lambda is translated to a class. One class is for the inner lambda. The free variable is in the field. It takes a parameter and applies it to the field. The other class has a method that feeds its parameter to the constructor of a new instance of the first class, making it the value of its field, just as applying a value to the outer lambda would make a closure for the inner lambda.

Translating another familiar lambda term, the Y combinator $\lambda f.(\lambda x.f(x\ x)\ \lambda x.f(x\ x))$, may illuminate the translation further:

```
cla0: visitor {
  f;
  (visitor -> ( f  ( acceptor   acceptor )))
}
cla1: visitor {
  f;
  (visitor -> ( f  ( acceptor   acceptor )))
}
cla2: visitor {
  (visitor -> ( new cla0[acceptor]
                new cla1[acceptor]))
}

 new cla2[]
```

Note that `cla0` and `cla1` are identical. This can then be hand-optimized to

```
cla1: visitor {
  f;
  (visitor -> ( f  ( acceptor   acceptor )))
}
cla2: visitor {
  (visitor -> ( new cla1[acceptor]
                new cla1[acceptor]))
}

 new cla2[]
```

We have not yet reasoned formally about the class unification and dead code elimination that can be done to shorten the program resulting from the translation.

## 2.2 Implementation in Peripaton

We now present the implementation of this encoding in Peripaton. First we need a few classes to serve as data structures representing the abstract syntax trees of the source language:

```
Goal : visitor {
  expr;
}
Abstraction : visitor {
  id;
  expr;
}
Application : visitor {
  expr1;
  expr2;
}
LambdaIdentifier : visitor {
  uid;
}
```

Since everything in Peripaton is a visitor, these classes also extend `visitor`. They are used only for data, however, and contain no functionality. `Goal` stands for the entire program, the root of the tree. Since a program in the lambda calculus is simply an expression, `Goal` has one field, which stands for an expression. An expression can be an abstraction, an application, or a variable. For these we have the classes `Abstraction`, `Application`, and `LambdaIdentifier` (the last is so named in order to distinguish it from an identifier in the target language). Peripaton is not typed, but if it were, the `id` field of `Abstraction` would be typed to be a `LambdaIdentifier`; it represents the bound variable in that lambda term, while `expr` represents the function body. `Application` has two fields representing the receiver and argument. Finally, `LambdaIdentifier` should have as its field a unique identifier as would be produced by `gensym`.

We need another set of visitor classes to build an abstract syntax tree in the target grammar:

```
TargetGoal : visitor {
  classList;
  expr;
}
Class : visitor {
  name;
  parent;
  fieldList;
  methodList;
}
Method : visitor {
  type;
  expr;
}
Invocation : visitor {
  expr1;
  expr2;
}
Creation : visitor {
  name;
  exprList;
}
AcceptorKW : visitor { }
VisitorKW : visitor { }
VisitorIdentifier : visitor {
  uid;
}
```

These are not conceptually different from the earlier set, but they reflect how a program in a Peripaton-like language

is a set of classes and a main expression; a class has a name, the name of a parent class, a set of fields, and a set of methods; and that a method is a dispatch type and a body. We also require classes `AcceptorKW` and `VisitorKW` to stand in for the keywords `acceptor` and `visitor`, since they can appear in the place of identifiers.

A visitor to perform the encoding will need methods to handle each source construct. It also needs to maintain a current variable (the one bound in the nearest enclosing abstraction) and a list of classes that have been generated so far. These will be passed as extra parameters. Consider the method for handling an application:

```
EncodingVisitor : visitor { ...
  (Application <v, cl> ->
     let t1 = (this {acceptor}.expr1 <v, cl>),
         t2 = (this {acceptor}.expr2
                <v, {t1}.classList>)
     in
       new TargetGoal[{t2}.classList,
                      new Invocation[{t1}.expr,
                      {t2}.expr]]
     end)
  ...
}
```

`Application` indicates the type of the parameter; this method will be called when a visitor of class `Application` or any descendent is applied to a visitor of class `EncodingVisitor` or any descendent, assuming no more specific methods have been written. `<v, cl>` list the extra, non-dispatching parameters which stand for the current variable and the list of classes so far.

The first step in the computation is to apply the visitor recursively to the first expression in the application. Since `acceptor` refers to the parameter (known to be of type `Application`), we extract the first expression using non-local field access, `{acceptor}.expr1`. We use `this` to apply the same visitor, passing extra arguments `v` and `cl` unchanged. This invocation returns a `TargetGoal`, which is useful not only to represent the finished product but also a current state in the translation process, as in this case. We store this in `t1`.

We want similarly to translate the other expression in the application. However, several more classes may have been added to the class list as a side effect of the translation of the first expression. We extract it from `t1` and pass it as an extra parameter in place of `cl`. We store the result in `t2`.

The return expression is a new `TargetGoal`. No classes are added at this point in the translation, but some may have been added in the translation of the second expression. Therefore we construct a `TargetGoal` with the `classList` from `t2` as its `cl` field. Its main expression, which conceptually has the same effect as the expression we are translating in the source language, is an invocation using the main expressions from the two `TargetGoal`s produced by the recursive applications of the visitor.

## 2.3 Removing non-local field references

We also use this example to build a bridge between the full Peripaton language and the core calculus we will use for formal reasoning later in the paper. In the rest of this section we will remove constructs from Peripaton by steps to arrive at the calculus, showing the code for translating `Application` at each step.

The first victim is non-local field references. These references make code concise but do not add to the expressive power of the language. Indeed, some philosophies of software engineering discourage non-local field reference, saying that fields ought to be retrieved by accessor methods. In lieu of non-local references, we must provide accessor methods for the classes that make up the abstract syntax tree. But what would these methods be like? Accessor methods typically take no parameters and have names similar to the fields they retrieve. Peripaton methods have no name and require at least one parameter.

To emulate this functionality we propose a programming idiom that uses accessor visitors— we create a new class that contains no functionality or data but is used only to trigger an accessor method. When we wish to retrieve the value of a field, we make an invocation with a visitor of the accessor class as the parameter. We can consider the parameter to be simply ignored— although in fact it is not ignored, since it is used for dispatching. In our example, we create a `GetExpr` class and a corresponding method for access to the `expr` field of `Goal`:

```
Goal : visitor {
  expr;
  (GetExpr -> expr)
}
GetExpr : visitor {}
```

To use this on a `Goal` stored in `g`, we write `(g new GetExpr[])`.

This idiom fits nicely with the object-oriented perspective of a method call being a "message send." An object of the getter class is a message; when a visitor visits that object, it responds to the message. Thus visitors can be compared to dynamic (or first-class) messages [10]. Observe the new code fragment:

```
EncodingVisitor : visitor { ...
  (Application <v, cl> ->
     let t1 = (this (acceptor new GetExpr1[])
                <v, cl>),
         t2 = (this (acceptor new GetExpr2[])
                    <v, (t1 new GetClassList[])>)
     in
       new TargetGoal[(t2 new GetClassList[]),
                      new Invocation[
                          (t1 new GetExpr[]),
                          (t2 new GetExpr[])]]
     end)
  ...
}
```

## 2.4 Removing non-dispatching arguments

Another language feature that is convenient but adds more headache than expressiveness when it comes to formal reasoning is the extra, non-dispatching arguments. To compensate for eliminating them from the language, we make the extra arguments into fields of the visiting class. When we wish to invoke the visitor recursively but supply new extra arguments, we replace `this` (which would use the old fields) with the creation of a new visitor, passing the extra arguments to the constructor to be used as fields. Since we will

not use `this` in the language from here on, we eliminate it also. A casualty of this is that class extension is less expressive, since `this` will always be replaced by an instance of the class defining that method rather than by a subclass on which this method may have been called. That does not come to play in our example. Observe the new code fragment:

```
EncodingVisitor : visitor { ...
  (Application ->
     let t1 = (new EncodingVisitor[v, cl]
                (acceptor new GetExpr1[])),
         t2 = (new EncodingVisitor[
                    v, (t1 new GetClassList[])]
                (acceptor new GetExpr2[]))
     in
       new TargetGoal[(t2 new GetClassList[]),
                      new Invocation[
                          (t1 new GetExpr[]),
                          (t2 new GetExpr[])]]
     end)
  ...
}
```

## 2.5 Removing let expressions

As a final step, we remove let expressions and non-field local variables. This is a simple matter of inlining:

```
EncodingVisitor : visitor { ...
  (Application ->
       new TargetGoal[
           ((new EncodingVisitor[
                 v,
                 ((new EncodingVisitor[v, cl]
                   (acceptor new GetExpr1[]))
                  new GetClassList[])]
             (acceptor new GetExpr2[]))
            new GetClassList[]),
           new Invocation[
               ((new EncodingVisitor[v, cl]
                 (acceptor new GetExpr1[]))
                new GetExpr[]),
               ((new EncodingVisitor[
                     v,
                     ((new EncodingVisitor[v, cl]
                       (acceptor new GetExpr1[]))
                      new GetClassList[])]
                 (acceptor new GetExpr2[]))
                new GetExpr[])]]
     )
  ...
}
```

As a final note, to attain a core calculus we also eliminate `gensym` and the unique identifiers it produces. This does not affect the fragment shown, but it is an important part in the full program.

# 3. THE VISITOR CALCULUS

## 3.1 Syntax and semantics

To describe the computation, we formally define values:

$$v \quad ::= \quad \texttt{new}\ t[\overline{v}] \qquad \textit{values}$$

We declare a pre-defined class:

$$\texttt{visitor}\ \{\ (\texttt{visitor} \mapsto \texttt{acceptor})\ \}$$

To describe the operational semantics, we use the notion of a context, i.e., an expression with a hole in it. The metavariable $X$ ranges over evaluation contexts, and the notation $X\langle e\rangle$ means context $X$ with its hole filled with $e$.

Operational Semantics:

$$\frac{\begin{array}{c}\textit{find-method}(\overline{c}, t_1, t_2) = (t \mapsto e) \\ \textit{fields}(\overline{c}, t_1) = \overline{x}\end{array}}{\begin{array}{c}(\overline{c}, X\langle(\texttt{new}\ t_1[\overline{v_1}]\ \texttt{new}\ t_2[\overline{v_2}])\rangle \rightarrow \\ (\overline{c}, X\langle[\texttt{acceptor}, \overline{x}/\texttt{new}\ t_2[\overline{v_2}], \overline{v_1}]e\rangle\end{array}} \tag{6}$$

Field lookup:

$$\frac{}{\textit{fields}(\overline{c}, \texttt{visitor}) = \emptyset} \tag{7}$$

$$\frac{\begin{array}{c}t : t'\{\overline{x};\ \overline{m}\} \in \overline{c} \\ \textit{fields}(\overline{c}, t') = \overline{y}\end{array}}{\textit{fields}(\overline{c}, t) = \overline{x}, \overline{y}} \tag{8}$$

Method lookup:

$$\frac{t_1 : t'\{\overline{x};\ \overline{m}\} \in \overline{c} \quad (t_2 \mapsto e) \in \overline{m}}{\textit{match}(\overline{c}, t_1, t_2) = (t_2 \mapsto e)} \tag{9}$$

$$\frac{\begin{array}{c}t_1 : t'\{\overline{x};\ \overline{m}\} \in \overline{c} \quad \forall(\tau \mapsto \eta) \in \overline{m} \\ \tau \neq t_2 \\ \textit{match}(\overline{c}, t', t_2) = (t'' \mapsto e)\end{array}}{\textit{match}(\overline{c}, t_1, t_2) = (t'' \mapsto e)} \tag{10}$$

$$\frac{t \neq \texttt{visitor}}{\textit{match}(\overline{c}, \texttt{visitor}, t) = \bullet} \tag{11}$$

$$\frac{\textit{match}(\overline{c}, t_1, t_2) = (t_2 \mapsto e)}{\textit{find-method}(\overline{c}, t_1, t_2) = (t_2 \mapsto e)} \tag{12}$$

$$\frac{\begin{array}{c}t_2 : t'\{\overline{x};\ \overline{m}\} \in \overline{c} \\ \textit{match}(\overline{c}, t_1, t_2) = \bullet \\ \textit{find-method}(\overline{c}, t_1, t') = (t'' \mapsto e)\end{array}}{\textit{find-method}(\overline{c}, t_1, t_2) = (t'' \mapsto e)} \tag{13}$$

We have been claiming that under the VisiCalc semantics no *message not understood* or *message ambiguous* error could ever occur. We now verify that claim. We use $t : t'$ to indicate that $t$ is a child class of $t'$.

LEMMA 3.1. *find-method will always return a method.*

We define the *depth* of two classes $t$ and $s$ in $\overline{c}$, $\Delta(\overline{c}, t, s)$ to be the $n$ from the proof of Lemma 3.1.

LEMMA 3.2. *match returns no more than one method.*

LEMMA 3.3. *match returns $\bullet$ iff it does not return a method.*

LEMMA 3.4. *find-method returns no more than one method.*

Finally, we reach this theorem from Lemmas 3.1 and 3.4.

THEOREM 3.5. *For every ordered pair of classes from a set of classes, find-method returns exactly one method.*

## 3.2 Type system

VisiCalc is "almost" untyped. We do not need types to statically check invocations, for example, because the semantics ensures an applicable method will always be found. We do need one thing to guarantee that a program will run correctly— every creation must have the appropriate number of arguments. Therefore VisiCalc does have a minimal type system, but it is enough to prove preservation and progress theorems. A typing environment is not a mapping, but a single type, the "current" type in the type derivation (or the environment can be empty, $\emptyset$). We use the metavariable $B$ to range over types and the empty set, where as $t$ ranges over types only. Typing rules for VisiCalc:

$$\frac{x \in \mathit{fields}(\overline{c}, t)}{\overline{c}, t \vdash x} \tag{14}$$

$$\frac{\overline{c}, B \vdash e_1 \quad \overline{c}, B \vdash e_2}{\overline{c}, B \vdash e_1\ e_2} \tag{15}$$

$$\overline{c}, t \vdash \texttt{acceptor} \tag{16}$$

$$\frac{\overline{c}, B \vdash \overline{e} \quad \mid \overline{e} \mid = \mid \mathit{fields}(\overline{c}, t) \mid}{\overline{c}, B \vdash \texttt{new}\ t[\overline{e}]} \tag{17}$$

$$\frac{\overline{c}, t \vdash e}{\overline{c}, t \vdash (t' \mapsto e)} \tag{18}$$

$$\frac{\overline{c}, t \vdash \overline{m}}{\overline{c} \vdash t : t'\{\overline{x};\ \overline{m}\}} \tag{19}$$

$$\frac{\overline{c}, \emptyset \vdash e \quad \forall c \in \overline{c} \quad \overline{c}, \emptyset \vdash c}{\overline{c} \vdash \overline{c}\ e} \tag{20}$$

Note that (17), if $\overline{e}$ is taken to be empty, implies that $\overline{c}, \emptyset \vdash \texttt{new}\ t[\,]$.

LEMMA 3.6. **[Term substitution]** *If* $\overline{x} = \mathit{fields}(\overline{c}, t)$, $\overline{c}, t \vdash e$, $\overline{c}, \emptyset \vdash \overline{d}$, *and* $\overline{c}, \emptyset \vdash f$, *then* $\overline{c}, \emptyset \vdash e\{\texttt{acceptor}, \overline{b} := f, \overline{d}\}$.

THEOREM 3.7. **[Preservation]** *If* $\overline{c}, \emptyset \vdash e$ *and* $(\overline{c}, k\langle e\rangle) \to (\overline{c}, k\langle e'\rangle)$ *then* $\overline{c}, \emptyset \vdash e'$.

THEOREM 3.8. **[Progress.]** *If* $e$ *is closed and* $\overline{c}, \emptyset \vdash e$, *then either* $e$ *is a value or there exists* $e'$ *such that* $(\overline{c}, e) \to (\overline{c}, e')$.

LEMMA 3.9. *If* $(\overline{c}, e) \to (\overline{c}, e')$ *and* $e$ *is closed, then* $e'$ *is closed.*

We say that an expression $e$ is *stuck* if it is not a value and there is no expression $e'$ such that $(\overline{c}, e) \to (\overline{c}, e')$. A program *goes wrong* if it evaluates to a stuck expression.

THEOREM 3.10. *Well-typed programs cannot go wrong.*

# 4. FROM VISITORS TO OBJECTS

We have demonstrated the connection between functions and visitors, specifically that a lambda term is translated into a new class and a creation of an instance of that class. We now link the visitor calculus with an object-oriented system.

## 4.1 Featherweight Java

As our target language, we use Featherweight Java [6] without casts. We have removed casts because they are not needed for the translation of the visitor calculus. A Featherweight Java program is a list of classes and a main expression. The syntax and semantics of Featherweight Java without casts is in Figure 2. Note that this last semantics rule requires that the parameters to a function call be values. This differs from the standard FJ definition, enforcing call-by-value semantics. The typing rules are as follows:

**Subtyping:**

$$\frac{}{\overline{CD} \vdash C \texttt{ <: } C} \tag{30}$$

$$\frac{\overline{CD} \vdash C \texttt{ <: } D \qquad \overline{CD} \vdash D \texttt{ <: } E}{\overline{CD} \vdash C \texttt{ <: } E} \tag{31}$$

$$\frac{\overline{CD}(C) = \texttt{class}\ C\ \texttt{extends}\ D\{\ldots\}}{\overline{CD} \vdash C \texttt{ <: } D} \tag{32}$$

**Expression Typing:**

$$\frac{}{\overline{CD}; Z \vdash x \in Z(x)} \tag{33}$$

$$\frac{\overline{CD}; Z \vdash \eta_0 \in C_0 \qquad \mathit{fields\text{-}fj}(\overline{CD}, C_0) = \overline{C}\ \overline{f}}{\overline{CD}; Z \vdash \eta_0.f \in C_i} \tag{34}$$

$$\frac{\mathit{fields\text{-}fj}(\overline{CD}, C) = \overline{D}\ \overline{f} \quad \overline{CD}; Z \vdash \overline{\eta} \in \overline{E} \quad \overline{CD} \vdash \overline{E} \texttt{ <: } \overline{D}}{\overline{CD}; Z \vdash \texttt{new}\ C(\overline{\eta}) \in C} \tag{35}$$

$$\frac{\begin{array}{l}\overline{CD}; Z \vdash \eta_0 \in C_0 \qquad \mathit{mtype}(\overline{CD}, C_0, m) = \overline{D} \to C \\ \overline{CD}; Z \vdash \overline{\eta} \in \overline{C} \qquad \overline{CD} \vdash \overline{C} \texttt{ <: } \overline{D}\end{array}}{\overline{CD}; Z \vdash \eta_0.m(\overline{\eta}) \in C} \tag{36}$$

**Method Typing:**

$$\frac{\begin{array}{l}\overline{CD}; \texttt{this} : C, \overline{x} : \overline{C} \vdash \eta_0 \in E_0 \\ \overline{CD} \vdash E_0 \texttt{ <: } C_0 \\ \overline{CD}(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{\ldots\} \\ \mathit{override}(\overline{CD}, D, m, \overline{C} \to C_0)\end{array}}{\overline{CD} \vdash C_0\ m(\overline{C}\ \overline{x})\ \{\texttt{return}\ \eta_0;\ \}\ \text{OK in}\ C} \tag{37}$$

**Class Typing:**

$$\frac{\begin{array}{l}\overline{CD} \vdash \overline{\mu}\ \text{OK in}\ C \qquad \mathit{fields\text{-}fj}(\overline{CD}, D) = \overline{D}\ \overline{g} \\ K = C(\overline{D}\ \overline{g},\ \overline{C}\ \overline{f})\{\ \texttt{super}\ (\overline{g});\ \texttt{this}.\overline{f} = \overline{f};\}\end{array}}{\overline{CD} \vdash \texttt{class}\ C\ \texttt{extends}\ D\{\overline{C}\ \overline{f};\ K\overline{\mu}\}\ \text{OK}} \tag{38}$$

**Program Typing:**

$$\frac{\overline{CD} \vdash \overline{CD}\ \text{OK} \qquad \overline{CD}; \vdash \eta \in C}{\vdash (\overline{CD}, \eta) \in C} \tag{39}$$

$$
\begin{array}{rcll}
P & ::= & (\overline{CD}, \eta) & \textit{programs} \\
\eta & ::= & & \textit{expressions} \\
 & & x & \quad \textit{variable references} \\
 & | & \eta.f & \quad \textit{field references} \\
 & | & \texttt{new } C(\overline{\eta}) & \quad \textit{creations} \\
 & | & \eta.m(\overline{\eta}) & \quad \textit{method calls} \\
CD & ::= & \texttt{class } C \texttt{ extends } C \ \{ \ \overline{C} \ \overline{f}; \ K \ \overline{M}\} & \textit{classes} \\
K & ::= & C(\overline{C} \ \overline{f}) \ \{ \ \texttt{super}(\overline{f}); \ \texttt{this}.\overline{f} = \overline{f}; \} & \textit{constructors} \\
\mu & ::= & C \ m(\overline{C} \ \overline{x}) \ \{ \ \texttt{return } \eta; \} & \textit{methods} \\
v & ::= & \texttt{new } C(\overline{v}) & \textit{values}
\end{array}
$$

Auxiliary definitions useful for operational semantics:
**Field Lookup:**

$$\frac{}{\textit{fields-fj}(\overline{\texttt{CD}}, \texttt{Object}) = \emptyset} \tag{21}$$

$$\frac{\overline{CD}(C) = \texttt{class } C \texttt{ extends } D \ \{\overline{C} \ \overline{f}; \ K \ \overline{\mu}\} \qquad \textit{fields-fj}(\overline{CD}, D) = \overline{D} \ \overline{g}}{\textit{fields-fj}(\overline{CD}, C) = \overline{D} \ \overline{g}, \overline{C} \ \overline{f}} \tag{22}$$

**Method Type Lookup:**

$$\frac{\overline{CD}(C) = \texttt{class } C \texttt{ extends } D \ \{\overline{C} \ \overline{f}; \ K \ \overline{\mu}\} \qquad B_0 \ m(\overline{B} \ \overline{x})\{ \ \texttt{return } \eta; \} \in \overline{\mu}}{\textit{mtype}(\overline{CD}, C, m) = \overline{B} \to B_0} \tag{23}$$

$$\frac{\overline{CD}(C) = \texttt{class } C \texttt{ extends } D\{ \ \overline{C} \ \overline{f}; \ K \ \overline{\mu}\} \qquad m \text{ not defined in } \overline{\mu}}{\textit{mtype}(\overline{CD}, C, m) = \textit{mtype}(\overline{CD}, D, m)} \tag{24}$$

**Method Body Lookup:**

$$\frac{\overline{CD}(C) = \texttt{class } C \texttt{ extends } D\{ \ \overline{Cf}; \ K \ \overline{\mu}\} \qquad B_0 \ m(\overline{Bx})\{ \ \textit{return } \eta; \} \in \overline{\mu}}{\textit{mbody}(\overline{CD}, C, m) = (\overline{x}, \eta)} \tag{25}$$

$$\frac{\overline{CD}(C) = \texttt{class } C \texttt{ extends } D \ \{\overline{C} \ \overline{f}; \ K \ \overline{\mu}\} \qquad m \ \text{ not defined in } \overline{\mu}}{\textit{mbody}(\overline{CD}, C, m) = \textit{mbody}(\overline{CD}, D, m)} \tag{26}$$

**Valid Method Overriding:**

$$\frac{\textit{mtype}(\overline{CD}, D, m) = \overline{D} \to D_0 \text{ implies } \overline{C} = \overline{D} \text{ and } C_0 = D_0}{\textit{override}(\overline{CD}, D, m, \overline{C} \to C_0)} \tag{27}$$

Operational semantics:

$$\frac{\textit{fields-fj}(\overline{CD}, C) = \overline{Cf}}{(\overline{CD}, X\langle \texttt{new } C(\overline{e}).f_i \rangle) \mapsto (\overline{CD}, X\langle e_i \rangle)} \tag{28}$$

$$\frac{\textit{mbody}(\overline{CD}, C, m) = (\overline{x}, e_0)}{(\overline{CD}, X\langle \texttt{new } C(\overline{e}).m(\overline{v}) \rangle) \mapsto (\overline{CD}, X\langle e_0\{\texttt{this}, \overline{x} := \texttt{new } C(\overline{e}), \overline{v}\} \rangle)} \tag{29}$$

**Figure 2: The syntax and semantics for Featherweight Java without casts.**

## 4.2 Translation from visitors to objects

We now present a translation from VisiCalc to Featherweight Java, see Figure 3. We use $[\![(\overline{c}, \ e)]\!]_{\texttt{V}}$ to stand for the FJ classes and main expression that result from translating the VC program $(\overline{c}, \ e)$. $[\![e]\!]_{\texttt{V}}$ stands for the translation of an expression, and similarly $[\![(t \mapsto e)]\!]_{\texttt{V}}$ stands for the translation of a method. $[\![c]\!]_{\texttt{V}}^{\overline{c}}$ stands for the translation of a class in the context of a list of classes, and this is extended for the entire list of classes in the expected way. Finally, $[\![c]\!]_{\texttt{vm}}$ creates a special method for a class; this is also extended for a list of classes.

At the simplest level, a field reference is translated to a field reference, a creation is translated into creation, etc. A class, moreover, is translated into a class. This implies a very close correlation between the two languages and an isomorphism between the two class hierarchies. Since VisiCalc has no non-local field references, the translated program will not either, even though Featherweight Java allows them. Thus all fields are translated as indirections from `this` in rule (40). Since all methods are translated into methods with one argument, named `acceptor`, rule (43) translates parameter references into parameter references, even though the Featherweight Java syntax does not distinguish them from

$$
\begin{array}{rcll}
[\![x]\!]_{\mathtt{v}} & = & \mathtt{this}.x & (40)\\
[\![\mathtt{new}\ t[\overline{e}]]\!]_{\mathtt{v}} & = & \mathtt{new}\ t([\![\overline{e}]\!]_{\mathtt{v}}) & (41)\\
[\![(e_1\ e_2)]\!]_{\mathtt{v}} & = & [\![e_2]\!]_{\mathtt{v}}.\mathtt{accept}([\![e_1]\!]_{\mathtt{v}}) & (42)\\
[\![\mathtt{acceptor}]\!]_{\mathtt{v}} & = & \mathtt{acceptor} & (43)\\
[\![(t \mapsto e)]\!]_{\mathtt{v}} & = & \mathtt{visitor\ visit\_}t(t\ \mathtt{acceptor})\{\mathtt{return}\ [\![e]\!]_{\mathtt{v}};\} & (44)\\
[\![t : t'\{\overline{x};\ \overline{m}\}]\!]_{\mathtt{v}}^{\overline{c}} & = & \mathtt{class}\ t\ \mathtt{extends}\ t'\ \{ & (45)\\
 & & \quad \mathtt{visitor}\ \overline{x};\\
 & & \quad t\ (\mathtt{visitor}\ \overline{x}, \mathit{fields}(\overline{c}, t'))\ \{\mathtt{super}(\overline{y});\ \mathtt{this}.\overline{x} = \overline{x};\}\\
 & & \quad \mathtt{visitor\ accept}(\mathtt{visitor}\ x)\ \{\ \mathtt{return}\ x.\mathtt{visit\_}t(\mathtt{this});\}\\
 & & \quad [\![\overline{m}]\!]_{\mathtt{v}}\\
 & & \}\\
[\![t : t'\{\overline{x};\ \overline{m}\}]\!]_{\mathtt{vm}} & = & \mathtt{visitor\ visit\_}t(\mathtt{visitor\ acceptor})\{\ \mathtt{return\ this.visit\_}t'(\mathtt{acceptor})\} & (46)\\
[\![(\overline{c}, e)]\!]_{\mathtt{v}} & = & \mathtt{class\ \ visitor\ extends\ Object}\ \{ & (47)\\
 & & \quad \mathtt{visitor}()\ \{\ \}\\
 & & \quad \mathtt{visitor\ accept}(\mathtt{visitor}\ x)\{\ \mathtt{return}\ x.\mathtt{visit\_visitor}(\mathtt{this});\}\\
 & & \quad \mathtt{visitor\ visit\_visitor}(\mathtt{Visitor\ acceptor})\ \{\ \mathtt{return\ acceptor};\}\\
 & & \quad [\![\overline{c}]\!]_{\mathtt{vm}}\\
 & & \}\\
 & & [\![\overline{c}]\!]_{\mathtt{v}}^{\overline{c}}\\
 & & [\![e]\!]_{\mathtt{v}}
\end{array}
$$

**Figure 3: Translation of the visitor calculus to Java**

field references.

The most interesting part of this translation is how double dispatch is implemented since Featherweight Java does not support it. As is common in the visitor pattern, calling a visit method is actually a handshake requiring two method calls, one call to the visit method and one to the accept method. The accept method is necessary to select the correct visit method in the visitor. Thus an invocation in VisiCalc is translated in rule (42) to the call of a method in the acceptor named `accept`. As we see in rule (45), each class has a method so named which is selected by Featherweight Java's single dispatching and which calls an appropriate visit method, named for that class; such a method is translated by rule (44). Featherweight Java's single dispatching will now select this method.

But what if no appropriate method exists in the visitor class or any of its ancestors? The VisiCalc semantics then seek an appropriate visit method for the parent class of the acceptor. To implement this, the `visitor` class, which tops the hierarchy, has a visit method for each class which will be selected by single dispatch if no other method is found. This method simply calls the visit method appropriate for the acceptor's parent class. This is shown in rules (46) and (47).

Let us revisit the example from section 2, where we translated the apply function, $\lambda f.\lambda x.fx$, to the visitor calculus. If we translate that visitor program to Java using the above translation, we get:

```
class visitor extends Object {
  visitor() {}
  visitor accept(visitor x) {
    return x.visit_visitor(this);
  }
  visitor visit_visitor(visitor acceptor) {
    return acceptor;
  }
  visitor visit_cla1(cla1 acceptor) {
    return this.visit_visitor(acceptor); }
  visitor visit_cla0(cla0 acceptor) {
    return this.visit_visitor(acceptor); }
}
class cla0 extends visitor {
  visitor f;
  cla0 (visitor f){ super(); this.f = f; }
  visitor accept(visitor x) {
    return x.visit_cla0(this);
  }
  visitor visit_visitor(visitor acceptor) {
    return acceptor.accept(f);
  }
}
class cla1 extends visitor {
  cla1 (){ super(); }
  visitor accept(visitor x) {
    return x.visit_cla1(this);
  }
  visitor visit_visitor(visitor acceptor) {
    return new cla0(acceptor);
  }
}

new cla1()
```

Notice that `visitor` has visit methods for each of the two classes. Class `cla0` represents $\lambda x.fx$, and class `cla1` represents $\lambda f.\lambda x.fx$. Class `cla0` has a visit method to apply its argument to its field, and class `cla1` has a visit method which constructs a new `cla0`.

## 4.3 Correctness

First, we need some concept of translating typing environments in VisiCalc to typing environments in Featherweight Java. An empty environment translates to an empty environment, and an environment containing a class translates to a mapping of the fields in that class each to class `visitor`:

$$[\![\emptyset]\!]_{\mathtt{v}} = \emptyset \qquad\qquad [\![t]\!]_{\mathtt{v}} = \mathit{fields}(t) : \mathtt{visitor}$$

LEMMA 4.1. *If* $\overline{c}, B \vdash e$ *then* $[\![\overline{c}]\!]^{\overline{c}}_{\mathtt{v}}[\![B]\!]_{\mathtt{v}} \vdash [\![e]\!]_{\mathtt{v}} : \mathtt{visitor}$.

LEMMA 4.2. *If* $\overline{c} \vdash c$ *then* $[\![\overline{c}]\!]^{\overline{c}}_{\mathtt{v}} \vdash [\![C]\!]_{\mathtt{v}}$ *OK*.

We reach the following theorem from Lemmas 4.1 and 4.2

THEOREM 4.3. **[Type Preservation.]** *If* $\overline{c} \vdash (\overline{c},\ e)$ *then* $\vdash [\![(\overline{c},\ e)]\!]_{\mathtt{v}}$ *OK*.

Our goal is to say that if a VisiCalc program execution takes a step to a new state, then the encoding of that program into Featherweight Java takes a step or steps to become the encoding of the resulting state on the VisiCalc side.

THEOREM 4.4. **[Behavior Preservation.]** *If* $p \rightarrow p'$ *then* $[\![p]\!]_{\mathtt{v}} \rightarrow^* [\![p']\!]_{\mathtt{v}}$.

# 5. CONCLUSION

Our results provide a foundation for visitor-oriented programming. Future work includes (1) proving the correctness of the translation from the $\lambda$-calculus to the visitor calculus, (2) developing more programming idioms for visitor-oriented programming, (3) developing a statically-typed version, relating it to a typed $\lambda$-calculus and Featherweight Java with casts, and (3) investigating a visitor calculus with symmetric double dispatching, including a type system for preventing *message ambiguous* errors, with inspiration from [1].

# REFERENCES


[1] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995.

[2] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, pages 130–145, 2000.

[3] William Cook. Object-oriented programming versus abstract data types. In *Proceedings of REX Workshop/School on the Foundations of Object-Oriented Languages*. Springer-Verlag (*LNCS* 489), 1990.

[4] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill Book Company, 1992.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. Technical Report MS-CIS-99-25, University of Pennsylvania, 1999.

[7] Shriram Krishamurthi, Matthias Felleisen, and Dan Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proceedings of ECOOP'98, ninth European Conference on Object-Oriented Programming*. Springer-Verlag (*LNCS* 1445), 1998.

[8] Gary T. Leavens and Todd D. Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA '98 Conference Proceedings*, pages 374–387, 1998.

[9] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303. Springer Verlag, 1999.

[10] Susumu Nishimura. Static typing for dynamic messages. In *Proceedings of the 25 ACM Symposium on Principle of Programming Languages POPL*, 1998.

[11] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.

[12] J. C. Reynolds. User-defined types and procedural data as complementary approaches to data abstraction. In S. A. Schuman, editor, *New Directions in Algorithmic Languages, IFIP Working Group 2.1 on Algol, INRIA*. MIT Press, 1975. Reprinted in D. Gries, editor, Programming Methodology, Springer-Verlag, 1978, and in C. A. Gunter and J. C. Mitchell, editors, Theoretical Aspects of Object-Oriented Programming.

[13] Scala homepage. http://lamp.epfl.ch/scala.

[14] Standard ML of New Jersey. http://www.smlnj.org.

[15] Philip Wadler. The expression problem. Circulated on a private mailing list, November 1998.