The Expression Problem
Gracefully

Andrew P. Black
Portland State University
The “expression problem” (EP) [38, 10, 46] is now a classical problem in programming languages. It refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants.

1 Introduction

The “expression problem” (EP) [38, 10, 46] is now a classical problem in programming languages. It refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants. Traditionally, the kinds of data abstraction found in functional languages can be extended with new operations, but adding new data variants is difficult. The traditional object-oriented approach to data abstraction facilitates adding new data variants (classes), while adding new operations is more difficult. The Visitor Pattern [13] is often used to allow operations to be added to object-oriented data abstractions, but the common approach to visitors prevents adding new classes. Extensible visitors can be created [43, 50, 31], but so far solutions in the literature require complex and unwieldy types, or advanced programming languages.

In this paper we present a new approach to the EP based on object algebras. An object algebra is a class that implements a generic abstract factory interface, which corresponds to a particular kind of algebraic signature [18]. Object
What is the Expression Problem?

- Consider a simple implementation of (immutable) lists

<table>
<thead>
<tr>
<th>Representations</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConsList (e, l)</td>
<td>first: return e, rest: return l, isEmpty: false</td>
</tr>
<tr>
<td>EmptyList</td>
<td>first: error, rest: error, isEmpty: true</td>
</tr>
</tbody>
</table>
Algebraic data types:

- Organize program by columns

<table>
<thead>
<tr>
<th>Representations</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConsList (e, l)</td>
<td>first: return e, rest: return l, isEmpty: false</td>
</tr>
<tr>
<td>EmptyList</td>
<td>first function: error, rest function: error, isEmpty function: true</td>
</tr>
</tbody>
</table>
Algebraic data types:

- Organize program by columns
  - easy to add a new column, but hard to add a new row

<table>
<thead>
<tr>
<th>Representations</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConsList (e, l)</td>
<td>first: return e, rest: return l, isEmpty: false</td>
</tr>
<tr>
<td>EmptyList</td>
<td>first function: error, rest function: error, isEmpty function: true</td>
</tr>
</tbody>
</table>
Objects

- Organize program by rows

<table>
<thead>
<tr>
<th>Representations</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConsList (e, l)</td>
<td>first: return e, rest: return l, isEmpty: false</td>
</tr>
<tr>
<td>EmptyList</td>
<td>first: error, rest: error, isEmpty: true</td>
</tr>
</tbody>
</table>

ConsList class

EmptyList class
Objects

- Organize program by rows
  - easy to add a new row, but hard to add a new column

<table>
<thead>
<tr>
<th>Representations</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConsList (e, l)</td>
<td>first</td>
</tr>
<tr>
<td></td>
<td>rest</td>
</tr>
<tr>
<td></td>
<td>isEmpty</td>
</tr>
<tr>
<td>return e</td>
<td>return l</td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
<tr>
<td>EmptyList</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>error</td>
</tr>
<tr>
<td></td>
<td>true</td>
</tr>
<tr>
<td>ConsList class</td>
<td></td>
</tr>
<tr>
<td>EmptyList class</td>
<td></td>
</tr>
</tbody>
</table>
Example: add an operation

- **One** new function with algebraic data, but **two** new methods in two classes with objects

<table>
<thead>
<tr>
<th>Representations</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConsList (e, l)</td>
<td>first: return e, rest: return l, isEmpty: false, print: &quot;[&quot;+e+...&quot;]</td>
</tr>
<tr>
<td>EmptyList</td>
<td>first: error, rest: error, isEmpty: true, print: &quot;[]&quot;</td>
</tr>
</tbody>
</table>

Monday, 6 July 2015
Why is this “difficult”?

• An editing problem
  - assumption: adding methods to two classes involves editing two files

• A packaging problem
  - assumption: the class is the smallest unit of modularity, so editing a class breaks modularity

• A typing problem
  - assumption: fields of the objects have been given types that allow just the base operations
Some History ...

Oliveira & Cook (ECOOP 2012):

“The “expression problem” (EP) [38, 10, 46] is now a classical problem in programming languages.”


46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list
Some History ...

Krishnamurthi et al. captured the issue:

“A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to
1. add a new variant of data and adjust the existing tools accordingly, and
2. extend the collection of tools.”


46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list
Some History ...

Krishnamurthi et al. captured the issue:

“A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to

1. add a new variant of data and adjust the existing tools accordingly, and

2. extend the collection of tools.”


46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list
Some History ...

Krishnamurthi et al. captured the issue:

“A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to

1. add a new variant of data and adjust the existing tools accordingly, and

2. extend the collection of tools.”


46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericty mailing list
Some History ...

Krishnamurthi et al. captured the issue:

“A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to

1. add a new variant of data and adjust the existing tools accordingly, and

2. extend the collection of tools.”

Restriction: “ideally, these extensions should not require any changes to existing code”

Wadler made this “problem” famous in 1998 (by coining a catchy name)

“The Expression Problem delineates a central tension in language design. Accordingly, it has been widely discussed, including Reynolds (1975), Cook (1990), and Krishnamurthi, Felleisen and Friedman (1998); the latter includes a more extensive list of references. It has also been discussed on this mailing list by Corky Cartwright and Kim Bruce. Yet I know of no widely-used language that solves The Expression Problem while satisfying the constraints of independent compilation and static typing.”


46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list
Wadler made this “problem” famous in 1998 (by coining a catchy name)

“The Expression Problem delineates a central tension in language design. Accordingly, it has been widely discussed, including Reynolds (1975), Cook (1990), and Krishnamurthi, Felleisen and Friedman (1998); the latter includes a more extensive list of references. It has also been discussed on this mailing list by Corky Cartwright and Kim Bruce. Yet I know of no widely-used language that solves The Expression Problem while satisfying the constraints of independent compilation and static typing.”


46. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list

Restrictions:
1. Static type safety (no casts)
2. No recompilation of existing code
Why wait until 1998?

- **Simula ’67, C++**
  - have algebraic data as well as objects

- **Smalltalk 80**
  - classes are not the unit of modularity

- **Visitor Pattern (name Visitor coined 1993)**
  - solves the problem, at the cost of pre-planning
Why wait until 1998?
Why wait until 1998?
Why wait until 1998?
Back to the future ...

• How was this “problem” solved in Smalltalk?
  - Classes named by global variables
  - *methods* are the unit of compilation & packaging
  - a package contains both *new classes* (and their methods) and extensions to existing classes (*new methods*)
  - loading a package into a Smalltalk system:
    ‣ *changes* some existing classes (overrides and adds methods, adds instance variables)
    ‣ introduces some new classes
symbol

^ self asParser token trimSpaces
Why does this work?

• Classes are (mutable) objects
  - adding (or changing) a method mutates the class

• Classes are named by global variables
  - loading a new version of a class definition changes the value of the global variable, and recompiles all existing methods

• Objects created by a methods in a class

• No modular type-checking
Why doesn’t it work in Java?

- Classes are *not* objects, and are immutable
  - Classes can be changed only by editing the source and recompiling
- Classes have global names, and cannot be renamed, assigned, or aliased
- Objects created by a language built-in `new`
- Modular type-checking
Java

e = new EmptyList
o = e.append(23)

Smalltalk

e := EmptyList new
o := e ++ 23
Java

```java
e = new EmptyList
o = e.append(23)
```

Smalltalk

```smalltalk
e := EmptyList new
o := e ++ 23
```

Data (row) extensibility is easy: add a new package defining a new class (but also must change creation code)
Java

```java
e = new EmptyList
o = e.append(23)
```

Smalltalk

```smalltalk
e := EmptyList new
o := e ++ 23
```

Data (row) extensibility is easy: add a new package defining a new class (but also must change creation code)

Operation (column) extensibility is impossible: can't change an existing class without editing the source.
• What about subclassing?
  - idea: subclass all of the original classes to create new variants with the additional operations.
  - Wadler focussed on generalizing the Java type system to make it possible to write those subclasses.

• But this doesn't help!
  - We still have to change all the creation code to use the new classes instead of the existing classes.
Grace

- new, simple O-O language
  - designed for teaching novice programmers the concepts of object-oriented programming
- block-structured within a module
- modules are objects
- no global variables
  - modules are imported under a name chosen by the client
Oliveira and Cook’s Example

<table>
<thead>
<tr>
<th>exp_base</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eval</td>
</tr>
<tr>
<td>Representations</td>
<td></td>
</tr>
<tr>
<td>lit(n)</td>
<td>n</td>
</tr>
<tr>
<td>sum(e₁, e₂)</td>
<td>e₁.eval + e₂.eval</td>
</tr>
</tbody>
</table>
Oliveira and Cook’s Example

<table>
<thead>
<tr>
<th>exp+pretty</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eval</td>
</tr>
<tr>
<td>lit(n)</td>
<td>n</td>
</tr>
<tr>
<td>sum(e₁, e₂)</td>
<td>e₁.eval + e₂.eval</td>
</tr>
</tbody>
</table>
dialect "staticTypes"

type Value = Object

type Exp = { eval -> Value }

factory method lit(i:Number) -> Exp {
    method x -> Number { i }
    method eval -> Value { x }
}

factory method sum(a:Exp, b:Exp) -> Exp {
    method l -> Exp { a }
    method r -> Exp { b }
    method eval -> Value { l.eval + r.eval }
}

// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour} = {threePlusFour.eval}"
// prints: an object = 7
Oliveira and Cook’s code from their paper, translated into Grace

```
dialect "staticTypes"

type Value = Object

type Exp = { eval -> Value }

factory method lit(i:Number) -> Exp {
  method x -> Number { i }
  method eval -> Value { x }
}

factory method sum(a:Exp, b:Exp) -> Exp {
  method l -> Exp { a }
  method r -> Exp { b }
  method eval -> Value { l.eval + r.eval }
}

// Demonstration:

def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour} = {threePlusFour.eval}"

// prints:  an object = 7
```

```bash
$ apbmg exp_base.grace
self.sum[0x0x7fc6cbc1b9f8] = 7
$```
Graceful solution

dialect "staticTypes"
import "exp_base" as baseExp
type Exp = baseExp.Exp & type { pretty -> String }

factory method lit(i:Number) -> Exp {
    inherits baseExp.lit(i)
    method pretty { x.asString }
}

factory method sum(a:Exp, b:Exp) -> Exp {
    inherits baseExp.sum(a, b)
    method pretty { "[{l.pretty} + {r.pretty}]" }
}

// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{{threePlusFour.pretty} = {threePlusFour.eval}}"
// prints: 3 + 4 = 7
Graceful solution

dialect "staticTypes"
import "exp_base" as baseExp
type Exp = baseExp.Exp & type { pretty -> String }

factory method lit(i:Number) -> Exp {
    inherits baseExp.lit(i)
    method pretty { x.asString }
}

factory method sum(a:Exp, b:Exp) -> Exp {
    inherits baseExp.sum(a, b)
    method pretty { "{l.pretty} + {r.pretty}" }
}

// Demonstration:
def threePlusFour:Exp = sum(lit 3, lit 4)
print "{threePlusFour.pretty} = {threePlusFour.eval}"
// prints:  3 + 4 = 7

$ apbmg exp+pretty.grace
self.add[0x0x7f9d40523c58] = 7
3 + 4 = 7
$
## Oliveira and Cook's Example

### Table: Representations and Operations

<table>
<thead>
<tr>
<th>Reprrsentations</th>
<th>exp and pretty</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eval</td>
<td>pretty</td>
</tr>
<tr>
<td>lit(n)</td>
<td>n</td>
<td>&quot;{n}&quot;</td>
</tr>
<tr>
<td>sum(e₁, e₂)</td>
<td>e₁.eval + e₂.eval</td>
<td>&quot;{e₁.pretty} + {e₂.pretty}&quot;</td>
</tr>
</tbody>
</table>
### Oliveira and Cook's Example

#### exp+pretty+bool

<table>
<thead>
<tr>
<th>Representations</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lit(n)</strong></td>
<td>eval: n; pretty: &quot;{n}&quot;</td>
</tr>
<tr>
<td><strong>sum(e₁, e₂)</strong></td>
<td>eval: e₁.eval + e₂.eval; pretty: &quot;{e₁.pretty} + {e₂.pretty}&quot;</td>
</tr>
<tr>
<td><strong>bool(b)</strong></td>
<td>eval: b; pretty: &quot;{b}&quot;</td>
</tr>
<tr>
<td><strong>iff(c, th, el)</strong></td>
<td>eval: if(c.eval) then {th.eval} else {el.eval}; pretty: &quot;if {c.pretty} then {th.pretty} else {el.pretty}&quot;</td>
</tr>
</tbody>
</table>

---

Monday, 6 July 2015


dialect "staticTypes"
import "exp_and_pretty" as baseExp

type Exp = baseExp.Exp

type Value = Object

method sum(l:Exp, r:Exp) -> Exp { baseExp.sum(l, r) }
method lit(x:Number) -> Exp { baseExp.lit(x) }

factory method bool(b:Boolean) -> Exp {
    method x -> Boolean { b }
    method eval -> Value { x }
    method pretty -> String { b.asString }
}

factory method iff(c:Exp, t:Exp, f:Exp) -> Exp {
    method eval -> Value {
        if (c.eval) then { t.eval } else { f.eval }
    }
    method pretty -> String {
        "if {{c.pretty}} then {t.pretty} else {f.pretty}"
    }
}

def e3plus4:Exp = sum(lit 3, lit 4)
def e2plus6:Exp = sum(lit 2, lit 6)
def ett:Exp = bool(true)
def ifExpr:Exp = iff(ett, e3plus4, e2plus6)

print "{ifExpr.pretty} = {ifExpr.eval}"

// prints: if (true) then 3 + 4 else 2 + 6 = 7
dialyct "staticTypes"
import "exp_and_pretty" as baseExp

type Exp = baseExp.Exp

type Value = Object

method sum(l:Exp, r:Exp) \rightarrow Exp 
{ baseExp.sum(l, r) }

method lit(x:Number) \rightarrow Exp 
{ baseExp.lit(x) }

factory method bool(b:Boolean) \rightarrow Exp 
{ 
    method x \rightarrow Boolean 
    { b } 
    method eval \rightarrow Value 
    { x } 
    method pretty \rightarrow String 
    { b.asString } 
}

factory method iff(c:Exp, t:Exp, f:Exp) \rightarrow Exp 
{ 
    method eval \rightarrow Value 
    { 
      if (c.eval) then { t.eval } else { f.eval } 
    }

    method pretty \rightarrow String 
    { 
      "if {{c.pretty}} then {t.pretty} else {f.pretty}" 
    }
}

def e3plus4:Exp = sum(lit 3, lit 4)
def e2plus6:Exp = sum(lit 2, lit 6)
def ett:Exp = bool(true)
def ifExpr:Exp = iff(ett, e3plus4, e2plus6)

print "{ifExpr.pretty} = {ifExpr.eval}"

// prints: if (true) then 3 + 4 else 2 + 6 = 7

...3 + 4 = 7
if (true) then 3 + 4 else 2 + 6 = 7

$
Object Algebras

- Oliveira and Cook. “Extensibility for the masses”. ECOOP 2012
  - Avoids typing issues (beyond type parameters) and permits re-use of creation code.
  - Basic idea: abstract over creation by defining a method that builds the structure on demand
  - Argument to that method is the “Object Algebra” — a factory object
dialect "staticTypes"
import "exp_base" as exp
type Exp = exp.Exp

// define the Object Algebra machinery
type IntAlg<A> = {
    lit(x:Number) -> A
    sum(e1:A, e2:A) -> A
}
factory method intFactory -> IntAlg<Exp> {
    method lit(x:Number) -> Exp { exp.lit(x) }
    method sum(a:Exp, b:Exp) -> Exp { exp.sum(a, b) }
}
method mk3Plus4<A>(v:IntAlg<A>) -> A {
    v.sum(v.lit(3), v.lit(4))
}

// compare the above with the normal expression:
// def e3Plus4:Exp = sum(lit 3, lit 4)
// add pretty—printing to expressions "retroactively"
type Pretty = { pretty -> String }
factory method prettyFactory -> IntAlg<Pretty> {
    factory method lit(x: Number) {
        method pretty -> String { x.asString }
    }
    factory method sum(a: Pretty, b: Pretty) {
        method pretty -> String { "{a.pretty} + {b.pretty}" }
    }
}

// demonstration
def x = mk3Plus4(intFactory)
// print "{x.pretty} = {x.eval}"
// fails: no method 'pretty' in object x
def s = mk3Plus4(prettyFactory)
// print "{s.pretty} = {s.eval}"
// fails: no method 'eval' in object s
print "{s.pretty} = {x.eval}"
// prints: 3 + 4 = 7
// add pretty—printing to expressions "retroactively"

```grace

// demonstration

def x = mk3Plus4(intFactory)
// print "\{x.pretty\} = \{x.eval\}"
// fails: no method 'pretty' in object x

def s = mk3Plus4(prettyFactory)
// print "\{s.pretty\} = \{s.eval\}"
// fails: no method 'eval' in object s

print "\{s.pretty\} = \{x.eval\}"
// prints: 3 + 4 = 7
```

```grace
3 + 4 = 7
```

```
null
```
Independent Extensibility

In real life, a much more common scenario than Fig. 1 followed by Fig. 2 followed by Fig. 3 would be like this. Some party $A$ defines $exp\_and\_pretty$. Another party $B$ independently defines $exp\_and\_bool$. A third party $C$ finds those and wants to combine them to $exp\_and\_pretty\_and\_bool$. This should be possible so that $C$ need only define pretty for bool (in addition to importing the two previous modules). Can Grace handle that?

- Adding $pretty$ uses inheritance, while adding $bool$ uses composition.
  - If both the original extensions used inheritance, we couldn't guarantee that we could combine them.
Independent Extensibility

In real life, a much more common scenario than Fig. 1 followed by Fig. 2 followed by Fig. 3 would be like this. Some party $A$ defines $exp\_and\_pretty$. Another party $B$ independently defines $exp\_and\_bool$. A third party $C$ finds those and wants to combine them to $exp\_and\_pretty\_and\_bool$. This should be possible so that $C$ need only define pretty for bool (in addition to importing the two previous modules). Can Grace handle that?

- Adding $pretty$ uses inheritance, while adding $bool$ uses composition.
  - If both the original extensions used inheritance, we couldn't guarantee that we could combine them

Yes!
- But solution is not fully general
Conclusions

• Wadler's version of the expression problem is unsolvable

• Wadler saw it as a challenge for type systems

• I see it as a challenge for even more fundamental features of a language:
  - global constants vs local namespaces
  - presence of built-in “non-objects”,
  - client object creation with method request or primitive