Functions and Traversals in Combination

Bryan Chadwick   Therapon Skotiniotis   Karl Lieberherr
Northeastern University
{chadwick, skotthe, lieber}@ccs.neu.edu

Abstract

In this paper we describe a new traversal organization inspired by ideas behind type-generic traversals that clearly separates traversal code and computation, implemented in an object-oriented setting. By delegating portions of the traversal and computation to separate functions we provide a library and methodology to develop succinct solutions to data-structure transformation problems. Because we provide an implementation of these new traversals via reflection in Java, all previous tools and libraries are still available and functionality can be extended and composed to deal with data structure reorganization using inheritance and encapsulation. We demonstrate the use of these abstractions by solving a few prototypical language problems: calculation of de Bruijn indices and type-checking.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms functional traversals, function objects, transformations

Keywords traversals, argument passing, reconstruction, transformation

1. Introduction

A common development scenario faced by programmers involves the implementation of operations over a rich, hierarchical data structure. Typical examples are found in language processing, and implementation, *e.g.*, source-to-source transformations, compilers, and XML libraries for adaptive object-oriented programming. These kinds of operations are usually done using visitors, hand-coded traversals, or domain specific languages, however, we can view such operations as functional transformations comprised of three parts:

traversal the order of sub-component traversal and the application of transformations over the data structure,

function(s) application specific tasks or transformations to be applied at each data type during traversal,

builder specialized combinations or selection of sub-components that produce a new data-structure.

Existing solutions admit limitations including duplicating traversal code and mixing traversal with functional and reconstruction code.[1] These limitations create solutions that are overly dependent on data structures making them less amenable to structural changes and difficult to extend. Furthermore, system constraints such as closed code libraries make existing solutions difficult to reuse.

To give us a specific data structure to discuss examples and extensions, we introduce a variation of the λ-calculus that includes first-class functions, integers, operations (addition and equality), and `if` expressions (Figure 1). Figure 2 defines our example language, in a mix of concrete and abstract syntax, as a Demeter style class dictionary [6]. Each name on the left side defines a data-type; ':' defines a *sum* type and '=' defines a *product* type. We include concrete syntax in quotations, interspersed with types, but exclude field names for brevity.

Our data structure is given in a language independent description, but similar data structures could be written in ML or Haskell. Ignoring issues related to the underlying programming language, consider the tasks related to implementing the following analysis and extensions to the example language as transformations, focusing on the three components: *traversals*, *functions*, and *builders*.

de Bruijn indices For our traversal we need to explore expressions in which variables may appear and expressions that can create new bindings. During traversal we need to maintain a list of bound variables and their corresponding lexical depth. At each expression that creates new bindings we need to add them to our list. Finally, our builder needs to construct new expressions with variable references replaced by their corresponding de Bruijn index.

Simple Type Checking Traversal needs to reach all expressions while maintaining a type environment. At each expression that defines new bindings we need to extend our type environment with the variables and their types. At each variable use we need to look up information in the type environment. While traversing all other expressions we must construct their types from sub-expressions; checking that none of our typing rules are violated in the process.

Lists Extending our language to support lists as values requires us to modify our data-structures, and extend our type checking solution. We should be able to easily update our traversal, function, and builder with just the new cases introduced by the addition of list related constructors, accessors and predicates.

All three extensions require information to be maintained during traversal. While calculating de Bruijn indices and type checking, information can be discarded after traversing binding forms. In the case of de Bruijn indices the function uses the address information collected to replace variable instances. When type-checking, the builder uses the environment to construct the type of each expression. Ideally, when adding list constructs, we should not have to modify any existing code, only extend it with cases for lists.

To support our design and test our claims, we provide a Java implementation consisting of a general traversal with default function and builder classes. Our traversal, implemented through Java reflection, visits each element of our data structure, allowing arguments to be passed along during traversal. We provide an extended dispatch strategy for methods defined in function objects (functions and builders). Our extended dispatch selects the most specific method definition considering all method arguments and

[1] We relate and compare our solution to previous work in Section 7.

$$
\begin{array}{lll}
e := n \mid x \mid op \mid (\texttt{if}\ e\ e\ e) & & \text{Expressions} \\
\quad \mid (\lambda\ (x_1 \cdots x_n)\ e) \mid (e\ e_1 \cdots e_n) & & \\
op := + \mid = & & \text{Operators} \\
v := n & & \text{Values}
\end{array}
$$

Figure 1. Concrete Syntax

```
E: Num | Var | Op | If | Lambda | Call.
Num = Integer.
Var: Sym.
Sym = Ident.
Op: Plus | Equals.
Plus = "+".
Equals = "=".
Arg = Sym.
If = "(if" E E E ")".
Lambda = "(lambda" "(" ArgList ")" E ")".
Call = "(" E EList ")".
```

Figure 2. Mixed Data Structure Syntax

```
Var: Sym | Addr.
Addr = Integer.


class AddrTrans extends IDf{
  Var apply(Var var, SymList senv){
    return new Addr(senv.lookup(var));
  }
}
```

Figure 3. Simple Address Replacement

TRAVERSAL

$$
\begin{array}{rcl}
t_{f,b}(d) & \Rightarrow & d' \\
& & \text{where } d' = f(d) \\
& & \quad d \text{ is atomic} \\
t_{f,b}(c(d_0,\cdots,d_n)) & \Rightarrow & f(b(c(d_0,\cdots,d_n),\ d'_0,\cdots,d'_n)) \\
& & \text{where } d'_i = t_{f,b}(d_i)
\end{array}
$$

DEFAULT FUNCTIONS

$$
\begin{array}{l}
id_f(d) \Rightarrow d \\
id_b(c(d_0,\cdots,d_n),d'_0,\cdots,d'_n) \Rightarrow c(d_0,\cdots,d_n) \\
b_c(c(d_0,\cdots,d_n),d'_0,\cdots,d'_n) \Rightarrow c(d'_0,\cdots,d'_n)
\end{array}
$$

Figure 4. Transformation Traversal Function

```
E: ... | Bool.
Bool: True | False.
True = "#t".
False = "#f".


class BoolTrans extends IDf{
  static E newtrue = Call.parse("(= 1 1)"),
  static E newfalse = Call.parse("(= 1 0)");

  E apply(True t){ return newtrue; }
  E apply(False t){ return newfalse; }
}
```

Figure 5. Boolean Literals and Rewrite Function

types. Default function and builder classes can be extended to specify operations on data types of interest.

Our code for the de Bruijn index calculation is shown in Figure 3 with the addition of a traversal argument shown in Figure 7 (discussed in Section 4). Our type checker in Figure 9 follows directly from the language's typing rules. Our type checker implementation for lists is a straight forward extension that only *adds* operations for the cases introduced to our language.

The rest of this paper is organized as follows: the next section describes a preliminary version of our general traversal, functions, and builders. Section 3 explains our extended dispatch mechanism. Section 4 extends our traversal with arguments and gives a detailed description of our de Bruijn indices calculation. Section 5 adds type annotations to our language and shows our simple type checker implementation as a builder. Section 6 extends our type checker implementation with lists. We discuss related work in Section 7, future work in Section 8, and conclude with Section 9.

2. Traversal and Recombination

To begin discussion of our traversal function and decomposition we first introduce a limited version that is useful for simple transformations. Figure 4 gives the specification of our new traversal function, parameterized by two other functions. f is a polymorphic function that takes a single argument and returns a result. A builder b is a function object that is responsible for the reconstruction of data types.

When d is atomic, *e.g.*, `int` or `boolean` (the first case of $t_{f,b}$), the builder function is not used; only f is called. When the data type being traversed is compound, signified by the use of a constructor c, we recursively traverse its data elements and reconstruct it using the builder, passing the original object as its first argument followed by the results of sub-component traversals. The resulting combination is then passed to f for transformation.

Below the definition of $t_{f,b}$ are definitions of useful instances of f and b. id_f is the identity function and id_b is the builder that ignores recomputed results, returning the original compound object. The builder b_c combines the recursive results using the same constructor, preserving the original types. We assume it is possible to inspect the data in some way to match arguments and aid in reconstruction. Though our Java implementation uses reflection to distinguish data types, languages with pattern matching can provide equivalent functionality.

To demonstrate one use of our new definitions, we add boolean constants to the example language, using them as abbreviations: $\#t$ will be short for the expression `(= 1 1)`, and $\#f$ for `(= 1 0)`. We will transform them using a newly defined function object and the predefined builder, b_c.

Figure 5 shows our additions to the syntax (both concrete and abstract) and the function object we use to transform boolean constants. The function when applied to `True` and `False` returns the appropriate `Call` expression. The static `parse` function knows how to create a `Call` given the string "`(= 1 1)`". The builder b_c will then reconstruct terms while traversing. The result will be a copy of the original term without boolean literals.

3. Dispatch and Matching

For the sake of this discussion we will refer to our Java implementation of *functions* as `Objects` which implement the correctly named methods [1].

Our library calls methods based on the expected name, the actual data types encountered during traversal, and all types in the defined method's signature. As shown in `BoolTrans` (Figure 5) functions are expected to implement the method `apply(·)` for

TRAVERSAL

$$t_{f,b,a}([d],\ d_a) \Rightarrow d' \quad \text{where } d' = f([d], d_a),\ d \text{ is atomic}$$

$$t_{f,b,a}(c(d_0,\cdots,d_n),\ d_a) \Rightarrow f(b(c(d_0,\cdots,d_n),\ d'_0,\cdots,d'_n), d'_a)$$
$$\text{where } d'_a = a(c(d_0,\cdots,d_n), d_a),\quad d'_i = t_{f,b,a}(d_i, d'_a)$$

DEFAULT FUNCTIONS

$$id_f(d, d_a) \Rightarrow d$$
$$id_a(d, d_a) \Rightarrow d_a$$
$$id_b(c(d_0,\cdots,d_n), d'_0,\cdots,d'_n, d'_a) \Rightarrow c(d_0,\cdots,d_n)$$
$$b_c(c(d_0,\cdots,d_n), d'_0,\cdots,d'_n, d'_a) \Rightarrow c(d'_0,\cdots,d'_n)$$

Figure 6. Argument Passing Traversal Functions

```
class SymExtender extends IDa{
  SymList update(Lambda lambda, SymList senv){
    return senv.push(lambda.formals);
  }
}
```

Figure 7. `Sym` Environment Extender

```
Arg = Type Sym.
Type: BoolT | IntT | FuncT.
BoolT = "bool".
IntT = "int".
FuncT = TypeList Type.
```

Figure 8. Added Syntax for Types

data types that they wish to transform. Similarly builders are expected to implement a `combine(·)` method.

Method calls are dispatched using a simple multiple dispatch matching algorithm that reflects on the formal and actual argument types at runtime. Method possibilities are explored until a match is found using a list of each argument's superclasses and interfaces. The method whose signature contains the exact types as the actual argument types is the most specific. The least specific is a method whose argument types are all `Object`. We choose the method which is the most specific for the first n arguments, where subclasses are always more specific than superclasses. The predefined builder b_c is implemented using the same lookup to find constructors of the original compound object's type. If no suitable method is found we throw an error, to help with traversal debugging.

The next section discusses an addition to the traversal function that makes it possible to pass arguments. We use the same dispatch strategy to allow modification of this parameter throughout traversal.

4. Traversal Arguments

Now that we have an idea of the basic traversal and our dispatch implementation, we make a simple addition that allows us to solve more interesting problems. For our second modification of the example language, we add support for de Bruijn indices to our abstract syntax. Figure 3 contains the syntax additions and the `AddrTrans` function object which extends id_f. We introduce `Addr`, an integer offset, as a variant of `Var`. We do not need to add concrete syntax, as `Addr` will only appear after rewriting.

We must keep track of a symbol environment (`SymList`); replacing each variable access with the index of its symbol in the current environment. To support these requirements, we need to parameterize the traversal with a third function object, an *augmentor*, and add an argument to the traversal and each of the function objects.

Figure 6 shows the modified traversal and default functions. Before recurring into a data element, we use the augmentor to calculate a new argument for sub-traversals. The augmentor's result becomes the last argument passed to the traversal and other functions. Our traversal implementation in Java expects the augmentor to implement a function, `update(·)`, as seen in Figure 7.

5. Type Checking

As a third addition to our mini-language we integrate type annotations and write a simple type checker using a combination of function objects. The modifications to the abstract and concrete syntax are shown in Figure 8. We add a tupe annotation to `Arg` representing the type of λ argument(s). In this new syntax, we can write the identity function for integers as: (λ ((**int** x)) x), while the *type* of `Plus` can be written as (`int int -> int`). We will only infer the type of the body of *lambda* expressions.

The problem of type checking an expression in this language is different from all of the previous examples in that the traversal is not *type-preserving*, *i.e.*, given an `E`, we want to return a `Type`, not another `E`. It is also not *type-unifying*, meaning we do not always return a `Type`, but sometimes return a `TypeList`. For example, when traversing a `Call` expression we build the type of the procedure and a list of actual argument types, inferred from the given expressions.

The solution here is to create a customized builder that will return a `Type` for expressions, but return a `TypeList` when we encounter an `ArgList`, or `EList`. As before we need to keep track of a type environment. Figure 9 contains a function object, `TypeBuilder`, that checks the types of expressions in our example language, and `TEnvExtender`, holds pairs of (`Sym`, `Type`). For simplicity we use a Lisp-style list representation for `ArgList`, `EList` and `TypeList`. We represent `ConsL` and `EmptyL` as `interfaces` and assume utility methods are implemented (*e.g.*, `equals(·)`). We use the exception `TE` to signal a type error.

One major benefit of our traversal organization is that our type checker implementation follows directly from the language's typing rules including the more complex cases of expressions, (**if** $e\ e\ e$) and ($e\ e_1 \cdots\ e_n$). As before, environment extension is encapsulated in a single function object (`TEnvExtender`). In the next section we reuse this `TypeBuilder` to add list types to our language.

6. Adding List Values

Given our organization of traversals, builders and arguments, we can extend our type checker to handle homogeneous lists by adding new syntax and simple rules for the types of various operators. Figure 10 shows the new syntax for lists and the simple additions to the `TypeBuilder`.

We add a new operator for each built-in function. Each operator requires the element type to be given, except `"empty?"`, which implicitly accepts the type `any`. Because our reference implementation is written in Java, we can use inheritance to reuse our earlier methods, adding only the specifics for this extension. We again use the static `parse(·)` function for brevity; constructing a `String` that represents the type of the operator. Notably our transformations `BoolTrans` and `AddrTrans` are unaffected by this change.

```
class TypeBuilder extends IDb{
// IF-expression
   Type combine(If ife, Type ttest, Type tthen, Type telse, TEnv te){
      if(!ttest.isBoolT())throw new TE("Condition Not Boolean");
      if(tthen.equals(telse))return telse;
      throw new TE("Then and Else not the Same Type");
   }
// Call-expression
   Type combine(Call call, Type proc, TypeList args, TEnv te){
      if(!proc.isFuncT())
         throw new TE("Args Applied to Non-func");
      FuncT f = (FuncT)proc;
      if(f.args.equals(args))return f.ret;
      throw new TE("Argument Types Incorrect");
   }
// Var-expression
   Type combine(Sym sym, Ident id, TEnv te)
    { return te.lookup(sym); }
// Lambda-expression
   Type combine(Lambda lam, TypeList args, Type ret, TEnv te)
    { return new FuncT(args, ret); }
// Op-expression
   Type combine(Plus plus, TEnv te)
    { return FuncT.parse("(int int -> int)"); }
   Type combine(Equal equal, TEnv te)
    { return FuncT.parse("(int int -> bool)")}

// Num, Arg-expression
   Type combine(Num num, Integer i, TEnv te){ return new IntT(); }
   Type combine(Arg arg, Type t, Sym sym, TEnv te){ return t; }

// EList and ArgList
   TypeList combine(ConsL cons, Type t, TypeList tlst, TEnv te)
     { return new TypeCons(t, tlst); }
   TypeList combine(EmptyL mt, TEnv te){ return new TypeEmpty(); }
}

class TEnvExtender extends IDa{
   TEnv visit(Lambda lam, TEnv te){
      return te.extend(lam.formals);
   }
}
```

Figure 9. Type Checker/Builder and TEnv Extender

Component	Method Name	Boolean Trans.	de Bruijn	Type Checking
function	apply()	id_f, Customized for Bool	id_f, Customized for Sym	id_f
builder	combine()	b_c	b_c	Custom for Type Rules
augmentor	update()	N/A	id_a, Customized for Var Env.	id_a, Customized for Type Env.

Table 1. Traversal Combination Summary

7. Related Work

Type-generic traversals (TGT) [5] are general functions that can be applied to terms of any type and allow recursive traversal into subterms. Transformations can be type-unifying (TU), were each recursive traversal returns the same type, or type-preserving (TP), were each recursive traversal returns the same type as it's input type. Information maintained during traversal and operations on them are encapsulated using a monad. Our augmentors and traversal argument serves the same purpose as the monad in TGT separating data from functionality. Our implementation traverses data structures in the same generic fashion as TGT, however, we separate traversal from construction so transformations can return any type. TU and TP transformations become special cases. While we give flexibility on return types checked dynamically, TGT provides statically typed TU and TP traversals.

The goal of Scrap Your Boilerplate (SYB) [2, 3, 4] is to automatically write code that traverses data structures, while the developer provides functions that perform transformations. Generic traversal functions take a combinator and specifies which of the nodes in the data structure a function should be applied to. The traversal combinator's argument is itself a function that transforms the data that it's interested in and acts like id for others. As with TGT, SYB provides both TU and TP traversals along with static type guarantees. Using SYB we believe it is possible to express some of the same transformations between TU and TP that are not possible in TGT, but we are not sure; that is a possible direction for future work.

The implementation of our generic traversal is similar to the WalkAbout class described by Palsberg and Jay [8]. We provide a similar mechanism, but separate functionality into multiple function objects, use multiple argument dispatch and a more involved matching algorithm.

Adaptive Programming (AP) [6] has provided different mechanisms for separating traversals from functionality. DemeterJ [9] trades dynamic control using a static, generative approach, while DJ [7] deploys a dynamic, refective traversal. Both use a domain specific language to describe a set of legal paths, which allows the

```
Op : ... | Cons | Empty
   | EmptyHuh | Car | Cdr.
Cons = "cons<" Type ">".
Empty = "empty<" Type ">".
EmptyHuh = "empty?".
Car = "car<" Type ">".
Cdr = "cdr<" Type ">".

Type: ... | AnyT | ListT.
AnyT = "any"
ListT = "[" Type "]".


class ListChecker extends TypeBuilder{
  Type combine(Car car, Type t, TEnv te)
   { return FuncT.parse("(["+t+"] ->"+t+")"); }
  Type combine(Cdr cdr, Type t, TEnv te)
   { return FuncT.parse("(["+t+"] -> ["+t+"])"); }
  Type combine(Empty mt, Type t, TEnv te)
   { return FuncT.parse("( -> ["+t+"])"); }
  Type combine(EmptyHuh mthuh, TEnv te)
   { return FuncT.parse("(any -> bool)"); }
  Type combine(Cons cons, Type t, TEnv te){
    return FuncT.parse("("+t+" ["+t+"]->["+t+"])");
  }
}
```

Figure 10. New Syntax and Type Checker for Lists

tools to optimize traversals according to the selected paths. Because traversal arguments and return values are fixed, combination and argument passing must be implemented through visitors via assignment. Our generic traversal has possibilities for more flexibility but is comparatively inefficient and lacks a form of traversal control.

8. Future Work

Our traversal abstractions have been refined from earlier ideas in both Adaptive Programming and Type-generic Traversals. We would like to continue exploring the uses of these abstractions in solving larger language problems related to interpreters and compilers, *e.g.*, XML or Java transformations within Eclipse.

Our first task will be the addition of some form of traversal control. In addition to the reference Java implementation, we would like to provide implementations in other languages. We would also like to add some form of static guarantees like those found in TGT and SYB.

9. Conclusion

We have described a functional decomposition of traversals that clearly separates traversal code, functional code, reconstruction, and traversal arguments. This separation allows us to build solutions to more complex problems while preserving the functional nature of transformations.

We provide an implementation of our ideas in Java that shows how function objects can be used to write transformations with or without traversal arguments. We can now solve problems like type checking that require a mix of type-preserving and type-unifying traversals. As a reference, Table 1 summarizes our function objects and interfaces; showing the different components used in each of our examples. We have also tested our traversal organization with promising results on other problems including structural duplication, Shannon Decomposition, and program evaluation.

References

[1] B. Chadwick. Traversal and library and code. http://www.ccs.neu.edu/home/chadwick/Traversal/, 2007.

[2] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[3] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

[4] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, Sept. 2005.

[5] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, Jan. 2002.

[6] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X.

[7] D. Orleans and K. J. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[8] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, Washington, DC, USA, 1998.

[9] The Demeter Group. The DemeterJ website. http://www.ccs.neu.edu/research/demeter, 2007.