Analysis and Transformation with the Nuthatch Tree-Walking Library

Anya Helene Bagge
University of Bergen, Norway
http://www.ii.uib.no/~anya/

Abstract

Nuthatch is a system for traversing, collecting information from, and rewriting trees, based on the idea of tree walking. The main application is software analysis and transformation. Nuthatch traversals are non-recursive by default and independent of the concrete tree representation. We provide an extensible library, Nuthatch/J, for doing tree walking in Java, with adapters for interfacing with popular software transformation tools like Stratego/XT and Rascal.

Transformations are described as walks that proceed in programmer-defined steps. Each step can perform actions based on observed properties of current node and walk, and affect state associated with the walk and also rewrite the walked tree. A step ends by walking to a different node in the tree, following the tree branches, and the walk ends by returning to the top.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.2.3 [*Coding Tools and Techniques*]: Object-oriented programming

General Terms Algorithms, Languages

Keywords Tree walking, tree traversal, pattern matching, transformation

1. Introduction

Popular program transformation frameworks, such as Stratego/XT [4], TXL [5], Tom [3] and Rascal [7] are based on an algebraic understanding of trees, where trees are terms, defined and manipulated recursively. In contrast, the Nuthatch[1] approach [2], treats trees as abstract data types, to be walked non-recursively by following the branches between the nodes. Branches can be followed in both directions; up to the parent, and down to the children.

The tree to the right illustrates the default walk, and options for deviating from it. The default walk will do a full traversal of the tree, visiting nodes depth-first in a left-to-right order.

The edge numbers show the order in which the nodes have been visited—note how the + node has been visited multiple times, at 4, 6 and 8.

*From the walker's current position, at *, we can directly visit +, 9 and f, with the default walk taking us to 9. Additionally, we may jump to an arbitrary node (say, 4).*

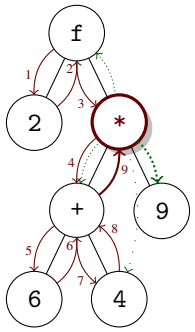

Figure 1. A nearly completed walk, following the default depth-first left-to-right order.

In Nuthatch, tree traversal is defined as a *walk* over a tree. A typical walk visits each node at least once. On a visit to a node, an *action* may be performed—such as gathering information, rewriting the tree, or changing the direction of the walk. Actions are typically chosen based on observation of the current state and current node, in a fashion similar to *join points* of aspect-orientation—for example, an action might be taken if the current subtree matches a given pattern, and we are moving upwards in the tree.

At any point in a walk, the *walker* is positioned at a particular node. The walker keeps track of which *branch* it entered the node from, either one of the children (numbered 1–n) or the parent (numbered 0). The incoming branch typically influences both the action to be performed, and which node to go to next.

In the default walk (see Figure 1), we visit every node of a tree, in depth-first left-to-right order. We select the next node to visit based on the previous node visited. For example, coming in from the parent (branch 0—see Figure 2), we will visit the leftmost child next (branch 1). Coming up from

[1] Named after the little bird (*Sitta* spp. [13]) famous for its ability to traverse trees in any direction.

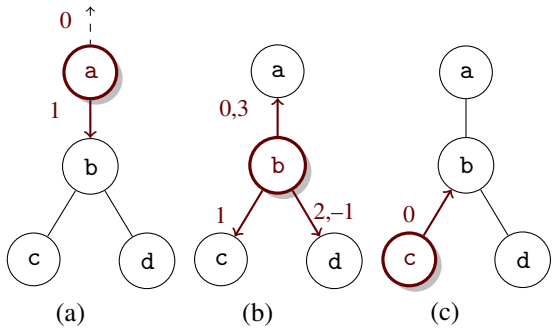

Figure 2. Numbering of outgoing branches. The current node (a, b or c) and valid branches are highlighted, with the canonical branch numbers indicated. Walking to the parent of the root node ends the walk, leaving the walker in the "top" position outside the tree. Subfigure (b) also shows the "wrap around" branch numbers, with $n+1 = 0$ and $-1 = n$, for an n-ary node.

child 2, we will go to child 3 next. After visiting the last child (or if there are no children), we go back to the parent—as is the case in the figure, where there is no child 3.

The Nuthatch/J library implements tree walking for Java, with support for multiple tree data structures, including those used by Rascal and Stratego/XT. Walks and actions are expressed as normal Java methods (usually in anonymous classes), and all normal Java features are available when implementing actions.

Contributions The Nuthatch approach was first introduced by Bagge and Lämmel [2]. While the previous paper describes the overall approach in terms of pseudo-code, this paper showcases the Java library that implements the approach and its use. Section 2 introduces the library, §3 shows some examples and usage scenarios for the approach, and §4 and §5 discusses motivation, implementation and experiences, and concludes.

2. The Nuthatch/J Library

Tree Cursors and Tree Handles Nuthatch abstracts over the concrete data structure used to implement trees, making it capable of interfacing with different systems. The currently supported tree types include Stratego/XT terms, Rascal algebraic data types and a simple built-in tree class.

Tree nodes may be typed, and may store a value and/or a (constructor) name. For example, in Figure 1, the leaves might have numbers as data values, while the operator names of the interior nodes are constructor names. The exact use of types, values and names depends on the tree data structure.

Each node is connected to its parent and children through *branches* (see Figure 2). A node of arity n has $n+1$ branches, numbered 0 for the parent branch, and $1 \ldots n$ for the n children. Additionally, we allow branch numbers to "wrap around" by one, for convenience, so that $-1 = n$ and $n + 1 = 0$.

getBranchHandle(i)	get tree handle of branch i
getCursor()	obtain cursor to node
getData()	get data, if any
getName()	get constructor name
getArity()	get number of children
getType()	get node type
hasBranch(i)	true if branch i exists
hasData()	true if node has data
hasName()	true if node has name
hasName(name)	true if node's name is *name*
hasType(type)	true if node has type *type*
isAtLeaf()	true if node is a leaf
subtreeEquals(other)	compare to other node
treeToString()	call toString() on handled tree
go(i)	move cursor to along branch
getFromBranch()	number of branch we came in from

Figure 3. The most commonly used methods on tree handles and tree cursors.

A *tree handle* abstracts over a tree data structure, providing methods to access its branches, data, name and type (see Figure 3). A *tree cursor* adds the notion of moving around in the tree, with a go method that changes the current node by following a branch. The position of a node in a tree is identified by its *path*—the branches one has to follow from the root to reach the node.

Handles and cursors are parameterized by the types of node types and node values. Typical methods for manipulating tree handles and cursors can be seen in Figure 3.

Walks and Walkers The overall traversal is implemented in the *tree walker*. The walker maintains a cursor to a place in a tree, and a *walk* object which specifies what should happen when a node is visited.

A *walk* object has a step method which is called every time a node is visited:

```
int step(W walker);
```

The step method can inspect the state of the walker—typically by examining the tree through the cursor—and returns a decision about where the walker should go next (a branch number).

A walker is initialized by giving it a tree to walk, and a walk to be walked. Once the walker is started, it will continue walking the tree until it is either stop, or it "falls off" the top of the tree.

Actions While a *walk* defines in which order the nodes are visited, an *action* defines what happens when a node is visited. Action objects are deceptively similar to walk objects, as they also define a step method:

```
1 int step(W walker);
```

The difference is that an action may also return a value of PROCEED instead of a particular next branch to be taken, leaving the decision to a containing walk object. Should

Meaning	Nuthatch/Java pattern
Matches anything	_
Matches node named n	*nodeName*(n)
Matches node with type t	*nodeType*(t)
Name n, children p0, ..., pk	*nodeWithChildren*(n, p1, ..., pk)
Matches/binds variable x	*var*("x") or just x (if declared)
Matches p, matches/binds x	*var*("x", p)
Matches both p1 and p2	*and*(p1, p2)
Matches either p1 or p2	*or*(p1, p2)
Doesn't match p	*not*(p)
Parent matches p	*parent*(p)
Ancestor matches p	*ancestor*(p)
Descendant matches p	*descendant*(p)
Node entered from branch b	*from*(b)

Figure 4. Supported patterns for matching, where n is a node name, p and p*i* are patterns, x is a variable t is a node type.

more control be necessary, the action can also return a particular branch number.

A typical use of an action is to wrap it in a default walk:

```
Walk<W> = new Default<W>(action); // W is the walker type
```

The *action factory* contains convenience methods for construction actions, particularly conditional actions and actions that involve pattern matching.

Join Points Actions are typically constrained so they occur only at particular *join points* in the walk. Common join points include:

- ◇ afterChild(w)—after the walker w has just visited a child
- ◇ beforeChild(w)—before the walker w will visit a child, in the default traversal order
- ◇ down(w)—walker w just came down a parent node
- ◇ up(w)—walker w will go up next, in default order
- ◇ leaf(w)—walker w is at a leaf node
- ◇ matches(w, pat)—current node matches pat

Pattern Matching The library also supports pattern matching. Nodes can be matched based on name, type, data value, children, parent, ancestors or any combination of these, as seen in Figure 4.

Patterns can have variables, with a binding environment controlled by the user. A bound variable will match only its value, while an unbound variable will match anything and become bound. Variables are easily combined with other patterns, so that one may conditionally bind a variable (e.g., only when a node has a certain name). Bound variables can also be used to refer to branches or subpaths to particular nodes.

Patterns are constructed through a pattern factory. Specialized pattern factories are available to deal with the peculiarities of the different tree backends.

A collection of static methods is also available, which in combination with Java's static import facility, can make patterns fairly readable:

```
// match any node named Foo
p1 = nodeName("Foo");
// any node named Bar, with first child named Foo,
// and second child matching/binding the variable x
p2 = nodeWithChildren("Bar", p1, var("x"));
// node named Bar, with parent named Foo
p3 = and(parent(nodeName("Foo")), nodeName("Bar"));
```

With a pattern factory specialized for the abstract syntax of the trees one is working on, it is possible to produce quite readable patterns. For example, the following matches a Define subtree, with either a FunClause or a ProcClause as the first child, and binds relevant variables:

```
Pattern<...> subcls = var("subcls"), name = ...;
Define(subcls, or(FunClause(name, ps, t),
                  ProcClause(name, ps))), mods, body)
```

The pattern objects come with match and build methods, that perform pattern matching and construct new trees by performing variable substitution on the pattern. Both methods require an environment with variable binding information. The pattern library comes with suitable scoped environment implementation. Building a new tree from a pattern also requires a BuildContext in order to construct values of the underlying data type.

As part of walking, pattern matching can be accessed through the walker, to match against the current node and to replace the current node. For example:

```
// if we're moving up the tree, replace 'x + 0' with 'x':
if (up(w) && w.match(Add(var("x"), Int(0))))
    w.replace(var("x"));
```

The walker maintains an environment which is local to the current step, but the user can also supply a specific environment.

Action Combinators So far, we have specified walks by writing step methods. For convenient composition of walks, we have a library of action combinators, accessed through an ActionFactory:

```
ActionFactory<Value, Type, TreeCursor<Value,Type>,
                  Walker<Value,Type>> af
    = FactoryFactory.getActionFactory();
```

With the action factory in hand, we can start composing actions:

```
Action<Walker<Value,Type>> action1 = ...;
Action<Walker<Value,Type>> action2 = af.up(action1);
```

This will make an action2 that performs action1 on the way up the tree. To create a walk (using the default traversal order) from an action, use af.walk:

```
Walk<Walker<Value,Type>> walk = af.walk(action2);
Walker<Value,Type> walker =
    new SimpleWalker<>(tree, walk);
walker.start();
```

As a more advanced example, the code below pretty-prints a tree in term form, with commas inserted between children:

```
toTerm = af.combine(
```

```
  af.atLeaf(appendData), // at leaf, print data
  af.down(appendName), // print name and paren going down
  af.up(appendEnd), // close parenthesis on the way up
  af.afterChild(af.beforeChild(appendComma))); // commas
```

In addition to normal actions, we also have MatchActions—actions that are to be performed after a match has happened. These actions are given an environment which contains bindings from the match. For example, `af.replace(var("x"))` creates a match action that replaces the current node with the result of substituting variables in the given pattern. Using this, we can implement the $x + 0 \rightarrow x$ example from the previous section like this:

```
af.up(af.match(Add(var("x"), Int(0)), af.replace(var("x"))))
```

The first parameter of af.match is a pattern to be matched, the second parameter is a match action to be taken if the pattern matched.

Complex match/replace actions can be built using a MatchBuilder, where multiple patterns and corresponding actions can be added in a manner similar to a case statement.

3. Examples & Usage Scenarios

Ancestor Matching Ancestor or parent pattern matching can be used in many cases where one would otherwise pass around an environment or use a symbol table—for example, if one needs to determine the name of the function the current tree node is located within, or even to look up a variable binding in a language with a let construct.

For instance, consider a tiny expression language with variables *Var*(n), integers *Int*(i), let bindings *Let*(v, e1, e2), addition *Add*(e1, e2) and multiplication *Mul*(e1, e2); where n is a string, i is an integer, e1 and e2 are expressions, and v is a variable expression.

The following pattern matches a variable (Var(...)), binds it to the meta-variable x, and also finds the closest ancestor matching the Let pattern with the same name x. The second argument of the Let is then bound to the meta-variable e.

```
and(Var(var("x")),
    ancestor(Let(Var(var("x")), var("e"), _)))
```

Attaching the pattern to code that prints the x and the e, we can test it on simple expressions Let(Var(y), Int(0), Var(y)) and get results like Var(y) -> Int(0).

Selective Traversal With a naive version of the match-and-print from the previous section, we will get into trouble, since only the variables that occur in the third child of a Let should actually be bound by it. In particular, since the first child is always a variable in our language and it has its own binding as a parent, it will always be printed.

To avoid this, we can attempt to control the tree traversal, so the first child of a Let is never visited. We can do this by simply saying that if we are just coming down into a Let, we should skip the first child and proceed immediately to the second. The following action accomplishes this:

```
af.match(Let(_, _, _), af.action(af.down(af.go(2))))
```

```
ActionBuilder<ExprWalker> as = af.sequenceBuilder();
// replace variable by its binding
as.add(af.match(and(Var(var("x")),
  ancestor(and(Let(Var(var("x")), var("e"),_), from(3)))),
 af.replace(var("e"))));
// skip first child of Let
as.add(af.down(af.match(Let(_,_,_),af.action(af.go(2)))));
// remove Lets on the way up
as.add(af.up(af.match(Let(_,_,var("e")),
  af.replace(var("e")))));
Action<ExprWalker> inline = as.done();
```

Listing 1. A variable inliner based on ancestor matching. The walk is built as a sequence of three actions to be tried at each step: (3) match and replace a variable with its binding, or (7) walk past the first child of a Let, or (9) remove Lets as we move up.

The af.go(...) action overrides the default walk—in this case only when we are entering the Let-node from its parent.

This kind of selective traversal is a fairly common scenario, and can be used, for instance, to exclude parts of a tree by redirecting the walk back to the parent when an undesirable nodes is encountered. A similar feature in Stratego and Rascal is the *topdown with stop* traversal.

Ancestor Matching Again If we study our variable binding printer more closely, we will see that it still does not behave correctly: Variables occurring in the second child of a Let (the value to be bound) will also seemingly be bound to the nearest Let. For example,

```
Let(Var(x), Int(1), Let(Var(x), Var(x), Var(x))):
  Var(x) -> Var(x)
  Var(x) -> Var(x)
```

where the latter two Var(x)s are both bound to the last Let.

We could try to solve this by excluding the second Let argument from traversal, but we are probably interested in seeing the binding of those variables as well. A better solution is to only consider Let expressions that occur as an ancestor of the third child:

```
ancestor(and(Let(Var(var("x")), var("e"), _), from(3)))
```

The from pattern will match if the current tree node was entered from the given branch number—the third child in this case. Using this technique gives the correct output,

```
Let(Var(x), Int(1), Let(Var(x), Var(x), Var(x))):
  Var(x) -> Int(1)
  Var(x) -> Var(x)
```

with the last Var(x) bound to the inner Let, and the second last is bound to the outer Let. Realistic code would probably evaluate and rewrite on the way, avoiding the confusing Var(x) -> Var(x) output at the end, where it is not clear that the two variables are distinct—an example of this can be seen in the variable inliner shown in Listing 1.

Evaluation-Order Walks The separation of action and walks means we can create custom walks that abstract away details of the language we are processing. For example, we

can create a walk that traverses a tree in the order of program execution, performing a user action along the way. If desired, the user action can override the default walk order by returning something other then PROCEED.

Let us construct an evaluation order walk for a simple statement language, based on the expression language of the previous examples, where we add assignment statements *Assign*(v, e), let statements *Declare*(v, e, s), conditions *If*(e, s1, s2), sequential execution Seq(s0,...,sn) and iteration *While*(e, s); where e is an expressions, v is a variable expression, and all s*i* are statements.

The overall evaluation order is bottom-up, with a given userAction applied to tree nodes in order of evaluation. Some cases need to be treated specially. We build the walk using action combinators, using a MatchBuilder to connect patterns for the special cases to actions:

```
XmplActionFactory af = XmplActionFactory.getInstance();
MatchBuilder<...> mb = af.matchBuilder();
```

For the case of Declare, and its expression counterpart Let, we would like to inspect it *after* the initializer expression has been visited. This corresponds to the point in time when the variable is declared and bound to its value. Also, we would like to visit after the body has been visited, corresponding to the time when the variable goes out of scope:

```
mb.add(Declare(_, _, _), af.combine(
  af.down(af.go(2)),  // skip first child on way down
  af.from(2, userAction),  // perform action after visiting init
  af.from(3, userAction)));  // also perform action after body
```

For the If statement, the condition is evaluated first. Then the user's action is presented with the If node, and can choose to go into either of the two branches (the default would be to visit the then-branch):

```
mb.add(If(_, _, _), af.combine(
  af.from(1, userAction),  // perform action after if-condition
  af.from(2, af.go(PARENT))));  // skip else-branch after then
```

This makes sense for evaluation scenarios—the exact traversal would depend on what we want to do; for data-flow analysis, we would likely want to visit both if-branches, and then execute some meet-action afterwards.

Similarly, the While is presented to the user action after visiting the condition, and then again after visiting the body. At this point, the default next step is up towards the parent, but the user's action can override this and visit the body again.

```
mb.add(While(_, _), af.combine(
  af.from(1, userAction),  // perform action after condition
  af.from(2, userAction)));  // perform action again after body
```

Other cases are handled similarly, with the default case being to visit expressions and statements on the way up.

Wrapping it up in a class, and adding it to our action factory gives us the evaluation-order walk as a combinator like any other. With a bit more work, it can be expanded to handle walking backwards, which would be useful in many kinds of analyses. A more sophisticated version would communicate extra information to the user action, in the same way that matching also passes an environment to the match action's step function. Constructs like goto can be handled by storing and jumping to the node path of labels. Further filtering and tuning, such as visiting expressions only, can be accomplished by creating additional custom combinators in combination with the ordered walk.

4. Discussion

Motivation The approach was originally motivated by a desire to reexamine advanced traversal strategies in an imperative context, and also to find ways of doing tree traversal for the Magnolia programming language, where an iterator/map-like style is preferred over recursion and loops.

The non-recursive approach has some advantages when working with large trees, such as AsFix2 trees in Stratego/XT [4] where recursion can lead to stack overflows.

The main benefit of the current Nuthatch/J library lies in the pattern matching facility, and the tree cursors that abstract over the tree representation. We have positive experience with this from the Magnolia compiler [1], where it also provides a significant speedup over the comparable Rascal code. Although the tree walking approach itself shows promise, it is difficult to tell if it superior to more standard techniques—particularly since the library does not have the same level of maturity as competing tools like Stratego/XT.

Implementation The core of the library lies in the *tree cursors*. A tree cursor is a pointer to the node of a tree, that also keeps track of the entire tree. Specific versions are bundled with the backend libraries, and additional implementations for another tree data structure (or any data structure that can be traversed in a tree-like fashion) can be added by the user.

In order to keep track of parents all the way back to the root of the tree, the cursor has a local stack of parent nodes, and also a *path* data structure, containing the path from the root to the current node (this is only necessary when the underlying tree structure is recursively defined, rather than providing parent navigation on its own—this is the case for all currently supported backends). Execution is iterative, so very deep trees can be traversed without adjusting the JVM stack. This also means that exceptions cannot be used to return to a previous point in the traversal, which is commonly done in recursive traversal. Pattern matching is mostly recursive, however, and has not been optimized (yet).

Nuthatch In Use The inversion-of-control style of Nuthatch can take some time to get used to. Rather than executing a series of calls and manipulating the result, a Nuthatch step function should "go with the flow", yielding control as the walk visits and processes children, and then picking up again when the walk comes back from the children. This also requires some assumptions about the traversal order of the default walk, though it also means that one can potentially abstract away from the walk order. Although it is possible to start recursive sub-walks, this can usually be avoided.

Although a notion of types is available for tree nodes, no type checking is currently done. As the library uses run-time composition of patterns and walks, any such checking would have to be done at run-time—static type safety would require a dedicated DSL (or embedding in a more advanced language than Java).

Beginners who are not well-versed in Java Generics can find the generic nature of Nuthatch a bit daunting. Many types have long lists of generic parameters, which are required in order to interface with different backend tree representation. There is no easy way to avoid this in Java—the best approach seems to be to provide specializations of the generic API for particular use cases, such as Rascal trees. Generating and maintaining these specializations can be somewhat of a hassle.

Points of Improvement The walker has, by default, no memory—unlike in a recursive approach, where the function retains it context when it returns from recursive calls. This requires some extra thought when designing steps. Often, enough of the context can be recovered by simply seeing which branch we entered from. Another approach is to store data in the walker, though we have tried to avoid this in recent versions of Nuthatch, preferring to let the user handle data through normal Java means.

5. Conclusion

Nuthatch is an approach to tree traversal, transformation and analysis, where the navigation (traversal) of the tree is separated from the actions to be taken when nodes are visited. Navigation is specified by walks that indicate the next node to visit in the tree. Visit actions are controlled through join points, that identify interesting points along the walk, and may also influence the navigation. The Nuthatch/J library implements the approach in Java, with support for pattern matching, join points, combinator-based building of actions, and multiple tree data structure backends, including Stratego terms and Rascal's PDB data types.

Nuthatch walks support generic traversal, in the sense of Stratego's strategies and Rascal's visit-statement, but is more general in the sense that at any point in the traversal, one can continue in any direction, including upwards in the tree, in a manner similar to zippers [6]. Tree walking bears resemblance to the visitor pattern [10], in particular advanced approaches with visitor combinators [9, 12] which support strategic traversal.

Nuthatch may be seen as an OO counterpart of parameterization or combinator-based traversal programming in functional programming [8, 11], but differs in the imperative stateful behavior, and the exposure of join points along the walks for customized traversal behavior.

Nuthatch is still at the experimental stage, but is used in the compiler for the Magnolia programming language [1], in performance-critical areas of the frontend. It works well as a Java library for pattern matching and tree traversal, though the action code tends to become verbose, with many inline anonymous classes. This will likely be better when we start using Java 8, with support for lambda expressions.

The source code of Nuthatch/J is available from GitHub at `https://github.com/nuthatchery/nuthatch`, and all examples in the paper are available from the web site at `http://nuthatchery.org/sle15/`.

Acknowledgments This research is partially financed by the Research Council of Norway, under the DMPL project. R. Lämmel, A. Hjortland and J.-P. Indrøy have contributed to the Nuthatch design. Comments from T. Hasu have been helpful in preparing this paper.

References

[1] A. H. Bagge. Facts, resources and the IDE/compiler mindmeld. In *4th Int'l Workshop on Academic Software Development Tools and Techniques (WASDeTT'13)*, July 2013.

[2] A. H. Bagge and R. Lämmel. Walk your tree any way you want. In *6th Int'l Conf on Model Transformation (ICMT'13)*, volume 7909 of *LNCS*, pages 33–49. Springer, June 2013.

[3] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking Rewriting on Java. In *18th Intl. Conf. on Term Rewriting and Applications (RTA'07)*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.

[5] J. R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[6] G. Huet. The Zipper. *Journal of Functional Programming*, 7:549–554, 9 1997.

[7] P. Klint, T. van der Storm, and J. J. Vinju. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *9th IEEE Intl. Working Conf. on Source Code Analysis and Manipulation (SCAM'09)*, pages 168–177. IEEE CS, 2009.

[8] R. Lämmel. The Sketch of a Polymorphic Symphony. In *Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *ENTCS*, pages 135–155, 2002.

[9] P.-E. Moreau and A. Reilles. Rules and Strategies in Java. In *Reduction Strategies in Rewriting and Programming (WRS'07)*, volume 204 of *ENTCS*, pages 71–82, 2008.

[10] J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. In *22nd Intl. Computer Software and Applications Conf. (COMPSAC '98)*, pages 9–15. IEEE CS, 1998.

[11] D. Ren and M. Erwig. A generic recursion toolbox for Haskell or: scrap your boilerplate systematically. In *ACM SIGPLAN Workshop on Haskell*, pages 13–24. ACM, 2006.

[12] J. Visser. Visitor combination and traversal control. In *OOPSLA '01: 16th ACM SIGPLAN Conf. on Object oriented programming, systems, languages, and applications*, pages 270–282. ACM, 2001.

[13] C. von Linné. *Systema Naturae*, volume 1. Salvius, Stockholm, Sweden, 10th edition, 1758.