Walk Your Tree Any Way You Want

Anya Helene Bagge[1] and Ralf Lämmel[2]

[1] Bergen Language Design Laboratory
Dept. of Informatics, University of Bergen, Norway
[2] Software Languages Team
University of Koblenz-Landau, Germany

Abstract. Software transformations in the NUTHATCH style are described as walks over trees (possibly graphs) that proceed in programmer-defined steps which may observe join points of the walk, may observe and affect state associated with the walk, may rewrite the walked tree, may contribute to a built tree, and must walk somewhere, typically along one branch or another. The approach blends well with OO programming. We have implemented the approach in the NUTHATCH/J library for Java.

1 Introduction

Software transformations rely fundamentally on traversing tree or graph structures, applying rules or computations to individual scopes, and composing intermediate results. This is equally true for model transformation (in the narrow sense), e.g., based on ATL [9] and for program transformation (including program generation and analysis), e.g., based on Rascal [13], Stratego [4], Tom [2], and TXL [6] as well as for less domain-specific programming models such as adaptive (OO) programming [19], generic (functional) programming [15], or OO programming with visitor combinators [30].

Transformation languages and programming models differ in how traversal is specified and controlled. For instance, in plain term rewriting with a hardwired normalization strategy such as innermost, traversal must be encoded in rewrite rules tangled up with the more interesting rules for primitive steps of transformation. By contrast, in Stratego-style programming [29, 30, 18] and some forms of generic functional programming [18, 15], schemes of traversal are programmer-definable abstractions that are parameterized in the rules or computations to be applied along the traversal, possibly tailored to specific nodes. For instance, consider this Stratego fragment for simplifying arithmetic expressions:

```
strategies
  simplify = bottomup(try(UnitLawAdd <+ ZeroLawMult))
rules
  UnitLawAdd : Add(x,0) -> x
  ZeroLawMult : Mult(x,0) -> 0
```

The library-defined traversal scheme *bottomup* is applied to rewrite rules for some laws of addition and multiplication. The programmer can reuse traversal schemes or define problem-specific ones, if needed.

In this paper, we describe a new transformation approach and a corresponding transformation language NUTHATCH,[3] which focuses programmer attention on the step-wise, possible state-accessing progression of a traversal, in fact, a *walk*, as opposed to the commitment to a traversal scheme and its application to rules. As an illustration, consider the following NUTHATCH fragment which matches the earlier Stratego example:

```
1 walk simplify {
2   if up then {
3     if ?Add(x, 0) then !x;
4     if ?Mult(x, 0) then !0;
5   }
6   walk to next;
7 }
```

The defined `walk` abstraction defines a complete walk over a tree. A walk starts at the root of the input term and (usually) ends there as well. In each step of the walk, a conditional statement is considered (line 2); it constrains rewrite rules (lines 3–4) to be applied when the walk goes `up` to the parent of the current node. Each rewrite rule consists of a match condition (see '?') and a replacement action (see '!'). The step is completed with a `walk to` statement (line 6) which defines the continuation of the walk. That is, the walk continues to the `next` node according to a default path for a comprehensive traversal.

Contributions

- We describe a notion of walks that proceed in programmer-defined steps which may observe join points of the walk, may access state associated with the walk, may rewrite the walked tree, may contribute to building a tree, and must walk somewhere, typically along one branch or another.
- We describe the realization of walks in the transformation language NUTHATCH. Conceptually, NUTHATCH draws insights from the concepts of tree automata [5], tree walking automata [1], continuations [24], and zippers [8]. Importantly, NUTHATCH incorporates state and supports OO-like reuse.
- We sketch NUTHATCH/J, an open-source library for walks in Java.[4]

The paper and accompanying material are available online.[5]

Road-map

§2 develops the basic notion of walks. §3 describes the NUTHATCH transformation language. §4 sketches the library-based implementation of NUTHATCH in Java. §5 discusses related work. §6 concludes the paper.

---

[3] Named after the nuthatch (*Sitta* spp.), a small passerine bird known for its ability to walk head-first towards the root of a tree, and on the underside of branches.
[4] http://nuthatchery.org/
[5] http://nuthatchery.org/icmt13/

2   The notion of walks

Walks walk along trees. Walks select branches. Walks complete paths. The default path is the starting point for all paths. Tree mutation may happen along the way.

2.1   Trees

In this paper, we mainly walk *trees*; graphs can also be walked as long as some distinguished entry node can replace the role of a root to reach all other nodes, also subject to precautions discussed in §3.10. In fact, we commit to *ordered trees*, i.e., trees with an ordering specified for the children. Ordered trees may be defined in two common ways, i.e., recursively (like terms of a term algebra) and graph-theoretically (with a designated root node and further constraints on nodes and edges for ordered trees as opposed to more general graphs). The graph-theoretical view is more helpful for intuitive understanding of walks.

We assume 'rich' trees in that nodes may be annotated with constructors and types (as needed for common term representations); leaves may carry some data (as needed for literals); edges (or '*branches*', as we will call them) may be annotated with labels (as needed for records, for example).

Thus, any node n of a tree t can be observed as follows:

- n.`arity`: The *arity* ≥ 0 of t's subtree rooted by n.
- n.`root`: Test for n being the root of t.
- n.`leaf`: Test for n being a leaf of t, i.e., n.`arity` $= 0$.
- n.`name`: The constructor name, if any, of n.
- n.`type`: The type, if any, of n.
- n.`data`: The data, if any, of n.
- n.`parent`: The parent node of n for n.`root` $=$ `false`.
- n.`child`$[i]$: The i-th child of n for $1 \leq i \leq n$.`arity`.
- n.`label`$[i]$: The label, if any, of the i-th child of n for $1 \leq i \leq n$.`arity`.
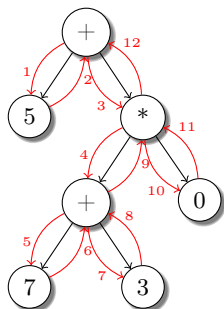
2.2   Branches

We limit ourselves to walks along the branches of trees as opposed to 'jumps', which would be possible in principle. This limitation seems to imply a more 'structured' programming technique. No need for jumps has arisen from our applications so far.

It is convenient to use natural numbers for referring to branches because 1, . . . , n.`arity` readily refer to the children of n, leaving 0 for the parent. Hence, it makes sense to use branch numbers to say that we walk to the parent or to a specific child. We may also use branches to track where we came from by referring to the 'previous node' with the corresponding branch number.

2.3   Paths

If we assume immutable trees for a moment, then the walk over a tree may be described as a *path*, i.e., sequence of nodes as they are encountered by the walk. Paths always start at the root of a tree. In the regular case, paths also end at the

The edge labels denote the order of walking along branches. The default walk combines pre-, in-, and post-order in that we walk down from the parent in a depth-first manner, and *we return to the parent after each subtree.*

Fig. 1. Illustration of the default path for an arithmetic expression.

root. Paths for walks along branches can be effectively represented as sequences of natural numbers.

We refer to the *default path* as the path which goes along each edge in the tree in both directions (i.e., along each branch) to achieve depth-first, left-to-right visiting order. Notably, a parent is visited before and after each child; see Fig. 1 for an illustration. The visiting order of the default path can be described by defining uniformly the `next` node (in fact, branch) relative to the `current` node and one of its branches, `from`:

$$\texttt{next} \mapsto \begin{cases} \texttt{from} + 1, \text{if } \texttt{current.arity} > \texttt{from} \\ 0, \text{otherwise} \end{cases}$$

We think of `from` as referring back to the node *from* which we walked to the `current` node. This is the information that needs to be tracked by a walk. That is, if we entered the `current` node from its parent (i.e., branch 0), then we walk to the first child; if we (re-) entered the `current` node from its i-th child, then we walk to the $i + 1$-th child, if there is a next child, and to the parent otherwise.

The definition of `next` is powerful in so far as it is also usefully describes continuation in 'default order', even for walks that diverted from the default path. This follows from the fact that the definition only looks at the branch to the immediately preceding node in the walk.

2.4   Join points

Walks (according to the default path or otherwise) expose 'join points' for transformations, i.e., the join points corresponding to the encounter of nodes along certain branches. Two important join points are described by these conditions on `current` and `from`:

– `down` $\equiv$ `from` $= 0$
– `up` $\equiv$ `current.leaf` $\|$ `from` $=$ `current.arity`

The `down` join point captures whether `current` was just entered from its parent. The `up` join point captures whether the walk is about to return to the parent of `current`. In §3 (see §3.6 specifically), we will see additional join points at work. Programmers quantify join points combined with other conditions on the tree and custom state to control the walk and to select stateful behavior.

2.5   Mutation

Let us consider walks on *mutable trees*. Thus, the steps of a walk may add and remove nodes and edges before they pick any branch. While a walk on an immutable tree is simply characterized by a sequence of contiguous branches, a walk on a mutable tree is characterized by a sequence of states. A *state s* has the following components:

- s.`tree`: The tree as seen in state s.
- s.`current`: The walk's current node in s.`tree`.
- s.`from`: The branch referring back to the node prior to s.`current`.

We assume that state transition breaks down into two components: the mutation of the `tree` and the actual step to advance `current`. Clearly, if we were to allow arbitrary mutation, the semantics of walking becomes totally operational and properties such as termination are no longer attainable.

  We are specifically interested in the case that mutation replaces `current` and its subtree, as in the application of a rewrite rule. When replacing `current`, though, the associated `from` may no longer be meaningful. Consider these cases:

- If `from` $= 0$, prior to mutation, then the first child, if any, of `current` was set up to be `next`. In this case, `from` shall be retained so that the first child, if any, of `current` is also set up to be `next` past mutation.
- If `current.arity` $> 0 \wedge$ `from` $=$ `current.arity`, prior to mutation, then the parent of `current` was set up to be `next`. Thus, `from` shall be assigned `current.arity`, as seen past mutation, so that again the parent of `current` is set up to be `next`.

These two cases cover rewrite rules on the `down` and `up` join points; for now, we take the view that `current` should not be replaced otherwise.

3   A Language for Walks

The NUTHATCH transformation language supports walks, as described in the previous section, on the grounds of an abstraction form for organizing walks in steps along branches. NUTHATCH can be mapped to an OO language such as Java, as discussed briefly in §4.

  At runtime, a walk encapsulates basic state, as described in §2.5, extra state to be declared, and it provides a step action to be invoked repeatedly. (We assume that walks are under the control of a main program which can start walks on trees, observe results after a walk is complete, and possibly restart suspended walks.)

3.1   Syntax summary

A walk abstraction has a name (an `id`), an optional declaration part for extra state associated with the walk and a statements part describing a step in terms of observing, matching, and rewriting the tree, accessing the walk's state and identifying the branch to follow. Walks may be parameterized, as discussed in §3.8. Thus:[6]

---

[6] We use ANTLR (`http://antlr.org/`) grammar notation.

```
walk : 'walk' closure ;
closure : id paras? '{' ('state' declaration)* statement+ '}' ;
paras : '(' id (',' id)* ')' ;
```

There are Java-like variable declarations, but with an optional type and a required initializer:

```
declaration : type? id '=' expression ';' ;
```

These are the available statement forms:

```
statement : '{' statement+ '}'
  | 'if' expression 'then' statement ('else' statement)?
  | declaration | id '=' expression ';' | expression ';'
  | 'return' expression ';'
  | 'walk' 'to' expression ';' | 'stop' ';' | 'suspend' ';'
  | '!' term ';'
  ;
```

Statement grouping, if-then-else with dangling else, (local) variable declarations, assignments, and expressions are Java-like. 'returns' are needed for functions; see below. There are special statement forms to specify what branch to `walk to`, to `stop` or `suspend` a walk. There is another special statement form to replace the current term (see '!').

In addition to Java-like expression forms, there are these special forms:

```
expression : ... | '?' term | getter | '~' id paras? ;
```

That is, there is a special expression form for matching the current term (see '?') in a condition that also binds variables. Further, there are 'getters' for trees (`arity`, `root`, etc.), the basic walk state (`tree`, `current`, `from`), join points (`down`, `up`), and `next`, as we set them up in §2. Tree observers are applied to the current term if not specified otherwise. The last expression form (see '~') deals with nested walks, as discussed in §3.9.

NUTHATCH also offers a simple abstraction form for *actions* which do not walk anywhere. Other than that, they can maintain state and observe the basic state of a walk in which they participate, if any. Likewise, there are *functions* for expression abstraction. Thus:

```
action : 'action' closure ;
function : 'function' closure ;
```

Actions and functions are illustrated in §3.8.

3.2   The Default Walk

The following NUTHATCH walk captures the default path of §2.3:

```
walk default {
  walk to next;
}
```

Each control-flow path of a NUTHATCH action must end in a walk-to statement which identifies the branch to walk to. The obvious options are `next`, `parent` (overloaded to refer to branch 0), `child[i]` (overloaded to refer to branch i), `first`

(assumed to represent the branch 1 for the first child), and `last` (assumed to represent the branch for the last child).

3.3 Diversion from the Default Path

The following example shows how a walk can be diverted depending on the current node; in this case, to avoid traversing `Expr` subtrees. To this end, we observe the `type` of the current node; we assume that `Expr` is one of the types of terms that are walked:

```
walk skipExpr {
  walk to (if type==Expr then parent else next);
}
```

(We use expression-level if-then-else.)

3.4 Derived Walks

New walks can be derived from existing walks. To this end, walk abstractions are referred to in statements. The underlying semantics is that the referenced walk's step action is inlined. For instance:

```
walk skipExpr {
  if type==Expr then walk to parent;
  default;
}
```

If the referenced walk includes extra state (which is not the case in the above example), then such state would be included into the referring walk automatically.

Because the default path is so prevailing, we assume that any walk abstraction derives implicitly from `default` such that `default`'s action is appended at the end of the step action. Accordingly, we shorten `skipExpr`:

```
walk skipExpr {
  if type==Expr then walk to parent;
}
```

We note that this implicit derivation occurs only at the top level, not when a walk is used to create a derived walk.

3.5 Stateful Walks

A walk may carry state. Actions may hence read and write such state. For instance, the following walk abstraction counts nodes; it takes advantage of the implicit derivation from `default`, as just explained above:

```
walk countNodes {
  state count = 0;
  if down then count++;
}
```

That is, we declare a variable `count` to maintain the node count, which we initialize to 0 and increment for each node, but only along the `down` join point—so that we do not count nodes multiple times. (We could also use `up` as a condition here.)

3.6   Flexible Point-cuts

We have started to invoke the AOP-like terminology of join points. Accordingly, walks may quantify the join points of interest; in AOP speak: walks need to express point-cuts. Consider the following walk abstraction which converts a tree into a string, using a term-like representation with prefix operators and comma-separated arguments as in "add(add(x,y),0)":

```
walk toString {
  state s = "";
  if leaf
    then s += data;
  else {
      if down then s += name + "(";
      if up then s += ")";
      if from>=first && from<last then s += ", ";
    }
}
```

In the code, we carefully observe the position along the walk to correctly parenthesize and place commas where appropriate. For instance, "(" belongs before the first child; thus the condition `down`, i.e., `from==parent`. This simple example clearly demonstrates how NUTHATCH style does not explicitly recurse / traverse into compound structures, as is the case with functional programming or Stratego-like traversal schemes. Instead, NUTHATCH style entails observation of the branch on which the current node was entered and possibly other data.

3.7   Walks with 'In Place' Rewriting

Rewriting is straightforward; it relies on a special condition form for use in an if-then-else statement to match ('?') a term pattern with the current term and to bind variables for use in the replace ('!') statement within the then-branch. We also say 'in place' rewriting to emphasize the fact that the tree is modified.

Let us revisit the example from the introduction (§1). The example follows the default path. When applied to the sample tree of Fig. 1, the result is '5'. For what it matters, we mention that simplification would not be complete, if we were using the `down` instead of the `up` join point in the example. (The unit law of addition would not be applicable in the example on the way down.)

'In place' rewriting is suitable for endogenous transformations [20] and specifically transformations that are meant to preserve many nodes and edges, as in the case of 'refining models' according to [26], but see §3.11 for a discussion of exogenous transformations [20].

3.8   Parameterized Walks

Common Stratego-like traversal schemes can be easily expressed by parameterizing walk abstractions, e.g.:

```
walk bottomup(s) { if up then s; }
```

The parameter `s` may abstract over actions such as rewrite rules. Let us revisit the example from the introduction (§1); we capture these actions (as of §3.1):

```
action UnitLawAdd { if ?Add(x, 0) then !x; }
action ZeroLawMult { if ?Mult(x, 0) then !0; }
action BothLaws { UnitLawAdd; ZeroLawMult; }
```

Thus, bottom-up traversal for simplification can be recomposed as follows:

```
bottomup(BothLaws)
```

Here is a more problem-specific, still language-parametric example of a parameterized walk which deals with state-based scope-tracking as opposed to Stratego-like traversal; such tracking is needed in various transformations, e.g., for the purpose of hosting new abstractions in the same context as the current scope or be it just for generating error messages.

```
walk scopeTracker(isDeclaration) {
  state scopes = new Stack[Node]();
  if down && isDeclaration then scopes.push(current);
  if up && current==scopes.top() then scopes.pop();
}
```

In the context of a transformation for Java, `isDeclaration` may be a condition (a function as of §3.1) that tests for a Java class declaration:

```
function isClassDec { return ?ClassDec(ClassDecHead(_,name,_,_,_),_); }
```

3.9   Nested Walks

Consider again the definition of `bottomup`, as given above. Now imagine that the argument `s` is not a plain action, such as rewrite rule, but it is meant to be a walk in itself. The existing definition would inline that walk according to the derivation semantics of §3.4, thereby disrupting the `bottomup` traversal. Instead, the argument walk should be performed atomically, as part of the referring step's action, as opposed to participating in the enclosing walk. References to arguments (which may be walks) can be accordingly marked as nested walks by '~':

```
walk topdown(s) { if down then ~s; }
walk bottomup(s) { if up then ~s; }
walk downup(s,t) { topdown(s); bottomup(t); }
```

('~' is a no-op on non-walks such as actions.) We note that each nested walk views the current node of the enclosing walk as the root. Note that no nested walk designation happens for `downup` because derivation semantics (as of §3.4) is appropriate here, if we want `s` to be applied on the way down and `t` on the way up. For comparison, consider these definitions:

```
walk badDownup1(s,t) { ~topdown(s); ~bottomup(t); }
action badDownup2(s,t) { ~topdown(s); ~bottomup(t); }
```

`badDownup1` performs a top-down walk followed by a bottom-up walk for each node in the tree. `badDownup2` performs a top-down walk followed by a bottom-up walk for a given tree; both walks start from the root.

3.10   Termination of Walks

A walk terminates regularly, if the walk encounters the root of a tree through the parent branch. A walk terminates irregularly if an unhandled exception is thrown by the step action. A walk may also be terminated explicitly or suspended via designated actions `stop` and `suspend`.

Accidentally, one may describe walks that do not terminate. This is implied by the expressiveness and flexibility of the abstraction form for walks. For instance, a transformation may continuously expand some redex for the `down` join point. Other programming techniques for traversals are also susceptible to this problem [16].

Another major challenge for termination is when graphs are walked. That is, walks may be cyclic. In adaptive programming [19], strategic programming on graphs [11], and OO programming with visitor combinator [30], this problem can arise as well. The problem can be solved, if we can make sure that no object is visited more than once. In NUTHATCH, we can use an 'enter once' walk as the starting point for any walk on a graph. Thus:

```
walk enteronce {
  state seen = new WeakHashSet();
  if down then
    if seen.contains(current)
      then walk to parent;
      else seen.add(current);
}
```

Thus, the walk keeps track of all nodes that were encountered. This scheme is not just useful for avoiding cyclic walks; it generally prevents walks from entering nodes more than once, even in directed acyclic graphs. The problem of non-termination or repeated walks into the same nodes can also be addressed if additional metamodel information is available to distinguish composition versus reference relationships, as in the case of walking EMF models, for example. That is, edges for reference relationships shall not be followed by walks.

3.11   Walks Building Terms

When facing exogenous transformations [20] (i.e., transformations with a target metamodel that is different from the source metamodel), then 'in place' rewriting (see §3.7) may not be appropriate, unless it is acceptable to operate on trees that use a 'union' metamodel for source and target models.

Suitable *tree builders* can be used to describe exogenous transformations or even endogenous transformations, when the source of the transformation is to be preserved. Consider the following walk that uses a tree builder to copy the walked tree, which is a good starting point for an endogenous transformation which preserves the walked tree:

```
walk copyall {
  state result = new TreeBuilder();
  if down then { result.add(current); result.moveDown(); }
  if up then result.moveUp();
}
```

The idea is that a tree builder provides an interface to (building) a tree; there are operations for adding nodes and edges. Further, the builder uses a *cursor* to maintain the current focus for addition. The cursor is a pointer to the children list of some node. Upon construction, the cursor points to the degenerated children list that will hold the root of the built tree. In the 'copy all' walk, we use the following operations:

– *add*: A given node (`current` in the example) is added to the children list pointed to by the cursor, where information such `name`, `type`, and `data` as well as `label` (for the edge to the parent) is copied over.
– *moveDown*: The cursor is set to point to the children list of the last node in the children list currently pointed to by the cursor.
– *moveUp*: The cursor is set to point to the children list of the parent node of the last node in the children list currently pointed to by the cursor.

When implementing exogenous transformations, tree builders are invoked to add 'terms' specific to the target model.

4 Walking in Java

In the following, we sketch the Nuthatch/J library for walking in Java. Nuthatch transformations can be mapped to Java code that uses the Nuthatch/J library.

4.1 Basic Interfaces

Nuthatch/J is designed as a generic tree walking library for Java which is independent of the underlying data representation. Thus, the library can be adapted by parameterization and subclassing for use with different kinds of trees, including those of existing transformation systems; see §4.4.

Walks are specified by implementing the `Walk` interface:

```java
public interface Walk<W extends Walker<?, ?>> {
    int step(W walker);
}
```

The `step` method performs a single step of the walk, can observe and manipulate state, and returns the next branch to `walk to`. The `Walker` type of the library encapsulates the tree-walking functionality and maintains the current node and state as described in §2.5, and provides the tree observers of §2.1.

The `Walk` interface is parameterized by the walker type, thereby making the extended features of a walker accessible in a type-safe manner. For example, the following code (also available online) implements the example from §1:

```java
public int step(ExprWalker w) {
  if (down(w)) {
    if (w.match(Add(var("x"), Int(0)))) w.replace(w.getEnv().get("x"));
    if (w.match(Mul(var("x"), Int(0)))) w.replace(Int(0));
  }
  return NEXT;
}
```

`ExprWalker` is a subtype of `Walker` which fixes the generics parameters for the expression terms of the example.

4.2 Extra State

Walk state is handled either by using variables in a closure or field variables in the class which implements `Walk`. The following Java code uses the former technique to replicate the example from §3.6:

```java
final StringBuffer s = new StringBuffer(); // Accumulate result here.
Walk<ExprWalker> toTerm = new BaseWalk<ExprWalker>() {
  public int step(ExprWalker w) {
    if (leaf(w)) // We are at a leaf; print data value.
      s.append(w.getData().toString());
    else if (down(w)) // First time we see this node; print constructor name.
      s.append(w.getName() + "(");
    else if (up(w)) // Just finished with children; close parenthesis.
      s.append(")");
    else // Coming up from a child (not the last); insert a comma.
      s.append(", ");
    return NEXT;
  }
};
```

4.3 Combinator Style

A library of common parameterized walk or action combinators (in the sense of §3.8) is available for various join points. In a combinator style, the simplifier of §1 can be expressed as follows:

```java
Walk<ExprWalker> w =
  walk(up(sequence(match(Add(var("x"), Int(0)), replace(var("x"))),
                   match(Mul(var("x"), Int(0)), replace(Int(0))))));
```

The walk is built up using static methods calls, where 'walk' represents the default walk, 'up' builds a conditional action for the up join point, 'sequence' executes all its arguments in the given order, 'match' executes its argument, if the pattern matches, and 'replace' performs a replace action.

4.4 Tool Integration

NUTHATCH/J integrates with Spoofax/Stratego/XT [4] and Rascal [13] so that these systems can be used in NUTHATCH/J applications. This is well in line with other transformation systems that support diverse access methods. For instance, Tom [21] can be applied to parse trees and object graphs of a domain model; POM adapters [12] allow Stratego to transform an Eclipse JDT AST.

The NUTHATCH/J+Stratego library supports untyped trees using the same term implementation as the Java version of Stratego. It also provides an interface to the JSGLR parser, including a pattern generator which generates pattern builders from an abstract syntax specification. Syntax definitions and minimal tooling for working on Java programs is also available, through the JavaFront package for Stratego.

The NUTHATCH/J+Rascal library wraps the Rascal data types into NUTHATCH trees, and can work on both concrete and abstract syntax trees (though without support for making concrete syntax patterns, at the time of writing).

	Nuthatch/J	Stratego	STRJ	Java
Collect Strings	3.0	5.0	4.2	—
Commute	4.6	29.8	0.9	0.8
Bottomup Build	5.6	3.2	1.2	—
Topdown	1.5	1.0	0.5	0.5
Downup	1.5	1.7	0.6	0.5

Table 1. Some performance measurements of Nuthatch/J vs. Stratego, with execution times in milliseconds (average over 5000 runs) for Nuthatch/J, interpreted Stratego, compiled Stratego (strj), and hand-written Java.

4.5   Performance

As of writing, Nuthatch/J has not yet been optimized for performance. Nevertheless, we have done some measurements of traversal and rewriting performance on Java programs, comparing against Stratego. All Nuthatch/J measurements were done using Stratego terms as the underlying data structure, so that we could use the exact same data for both Nuthatch/J and Stratego, and check that both implementations gave the exact same results.[7]

For reference, we also measured hand-written Java versions of some of the transformations, in order to get an idea of the top performance possible using the Stratego term library.

A few selected experiments are summarized in Table 1.[8] The experiments show that performance of Nuthatch/J is similar to that of the Stratego interpreter for trivial traversals (*topdown, downup*), but slower than compiled Stratego code. Simple transformations (*commute*) are a lot faster in Nuthatch/J than with interpreted Stratego code, but again, compiled Stratego is faster. Nuthatch/J has an advantage when using plain Java to accumulate state, and outperforms compiled Stratego on collecting strings from a tree.

5   Related Work

Walks à la Nuthatch combine generic traversal, stateful behavior, OO-like derivation, and parameterization. Accordingly, walks relate to Stratego-like programming, visitor programming including visitor combinators, adaptive programming, generic functional programming, and model transformation.

Stratego et al. Walks are inspired by the seminal work on strategies à la Stratego [29, 4]—the combination of term rewriting and programmable strategies, also for traversal purposes. Walks depart from strategies in that the basic traversal expressiveness is about continuous walking along branches as opposed to recursive

---

[7] Stratego measurements were done using both interpreted and compiled code, both using version 1.1 of the Spoofax language workbench. For interpretation, we used the *hybrid interpreter*, which uses compiled-to-Java versions of the standard libraries, but interprets user code on the fly. Measurements are an average of 5000 iterations, run on an otherwise idle AMD FX-8350 computer, running OpenJDK 7u15.

[8] See `http://nuthatchery.org/icmt13/benchmarks.html` for more details.

one-layer traversal. Further, walks are designed around state, whereas strategies only uses state in the special sense of dynamic rewrite rules [3]. Also, walks are designed to be derivable (and parameterized), whereas strategies leverage parameterization only. §3.8 shows how walks represent Stratego-like traversal schemes. The AspectStratego [10] variation on Stratego was proposed to leverage some means of aspect orientation in the context of term rewriting. In this work, join points of rewriting or the strategic program can be intercepted. By contrast, walks à la NUTHATCH interact with join points for walks along trees.

Visitor programming In the OO programming context, traversal problems can be addressed by means of visitors [22]. Specifically, advanced approaches use visitor combinators [30, 21] inspired by Stratego. The cited approaches transpose Stratego style to an OO language context; they make limited use of OO-like derivation and imperative state. When compared to walks, 'visits' are controlled strategically (as above), as opposed to exposing join points of the walks to the problem-specific functionality.

Adaptive programming The notion of processing object graphs in a structure-shy fashion has been realized in seminal work on adaptive programming [19], where traversal specifications of objects to be visited are separated from actions to be actually applied to the objects on the path. Stratego-like strategic programming and adaptive programming are known to be related in a non-trivial manner [17]. Walks differ from adaptive programs in that they do not leverage any special language constructs for traversal specifications. Also, each step of a walk may affect the remaining path.

Generic functional programming The parameterization- or combinator-based approach of traversal programming has been pushed particularly far in a generic functional programming context; see, e.g., the 'mother of traversal' [14, 23]. Indeed, such approaches offer highly parameterized abstractions for different traversal instantiations. By contrast, walks à la NUTHATCH additionally offer i) OO-like derivation, ii) imperative OO-like stateful behavior, and iii) exposure of join points of walks (traversals) for customized traversal behavior.

Model transformation Because of the large amount MT languages in existence, it is hard to compile a useful comparison. Overall, NUTHATCH style is closer to term rewriting approaches. We have in mind ATL [9] as a representative in what follows. Thus, model transformations match source model elements and map them to target model elements. Endogenous transformations, specifically, may rely on some degree of implicit behavior (refinement) to copy or retain model elements when not said otherwise [26]. MT rules are essentially declarative, with some built-in scheme of applying rules to the source model. Escapes to imperative features are needed in practice and thus supported. Join points of walks à la NUTHATCH are not established for MT languages.

6   Concluding remarks

We have described a new approach to traversal programming with walks as the central abstraction form. The development of the walk notion and all of our

related experiments were based on the NUTHATCH/J library for walks in Java. The NUTHATCH transformation language should be viewed as an ongoing effort to extract a transformation DSL from the NUTHATCH/J library. NUTHATCH can express traversal schemes à la Stratego and thus, it provides 'proven expressiveness'. Importantly, OO idioms (such as state, encapsulation, closures, and type derivation) are also part of the NUTHATCH programming model. The NUTHATCH/J library leverages adapters for tree formats of other transformation tools in the interest of tool integration.

Proper DSL notation enables conciseness (when compared to Java), type checking, static analyses for other properties of walks, and compile-time optimizations. However, an external DSL approach makes it harder to provide all language services. Therefore, we continue research on the NUTHATCH/J's combinator style of §4.3 to perhaps settle on an internal DSL (in fact, DSL embedding) which is a popular approach for transformation languages with functional host languages [18, 25, 7]. NUTHATCH/J's combinator style would also permit on-the-fly optimization, as it has been used elsewhere for embedded DSL implementation [27, 28].

References

1. Aho, A.V., Ullman, J.D.: Translations on a Context-Free Grammar. Information and Control 19(5), 439–475 (1971)
2. Balland, E., Brauner, P., Kopetz, R., Moreau, P.E., Reilles, A.: Tom: Piggybacking Rewriting on Java. In: 18th Intl. Conf. on Term Rewriting and Applications (RTA'07). LNCS, vol. 4533, pp. 36–47. Springer (2007)
3. Bravenboer, M., van Dam, A., Olmos, K., Visser, E.: Program Transformation with Scoped Dynamic Rewrite Rules. Fundamenta Informaticae 69(1-2), 123–178 (2006)
4. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. 72(1-2), 52–70 (2008)
5. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata` (2007), release October, 12th 2007
6. Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. 61(3), 190–210 (2006)
7. George, L., Wider, A., Scheidgen, M.: Type-Safe Model Transformation Languages as Internal DSLs in Scala. In: Theory and Practice of Model Transformations (ICMT 2012). LNCS, vol. 7307, pp. 160–175. Springer (2012)
8. Huet, G.: The Zipper. J. Funct. Program. 7(5), 549–554 (1997)
9. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
10. Kalleberg, K.T., Visser, E.: Combining Aspect-Oriented and Strategic Programming. In: Workshop on Rule-Based Programming (RULE '05). ENTCS, vol. 147, pp. 5–30 (2006)
11. Kalleberg, K.T., Visser, E.: Strategic Graph Rewriting: Transforming and Traversing Terms with References. In: 6th Intl. Workshop on Reduction Strategies in Rewriting and Programming (WRS'06) (2006), online publication

12. Kalleberg, K.T., Visser, E.: Fusing a Transformation Language with an Open Compiler. In: 7th Workshop on Language Descriptions, Tools and Applications (LDTA '07). pp. 18–31. ENTCS, Elsevier (2007)
13. Klint, P., van der Storm, T., Vinju, J.J.: 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). pp. 168–177. IEEE CS (2009)
14. Lämmel, R.: The Sketch of a Polymorphic Symphony. In: Reduction Strategies in Rewriting and Programming (WRS 2002). ENTCS, vol. 70, pp. 135–155 (2002)
15. Lämmel, R., Peyton Jones, S.L.: Scrap your boilerplate: a practical design pattern for generic programming. In: ACM SIGPLAN Intl. Workshop on Types in Languages Design and Implementation (TLDI'03). pp. 26–37. ACM (2003)
16. Lämmel, R., Thompson, S., Kaiser, M.: Programming errors in traversal programs over structured data. Sci. Comput. Program. (2012), in press. DOI: 10.1016/j.scico.2011.11.006
17. Lämmel, R., Visser, E., Visser, J.: Strategic programming meets adaptive programming. In: 2nd Intl. Conf. on Aspect-Oriented Software Development (AOSD'03). pp. 168–177 (2003)
18. Lämmel, R., Visser, J.: A Strafunski Application Letter. In: 5th Intl. Symp. on Practical Aspects of Declarative Languages (PADL'03). LNCS, vol. 2562, pp. 357–375. Springer (2003)
19. Lieberherr, K.J., Patt-Shamir, B., Orleans, D.: Traversals of object structures: Specification and Efficient Implementation. ACM Transactions on Programming Languages and Systems 26(2), 370–412 (2004)
20. Mens, T., Van Gorp, P.: A taxonomy of model transformation. ENTCS 152, 125–142 (2006)
21. Moreau, P.E., Reilles, A.: Rules and Strategies in Java. In: Reduction Strategies in Rewriting and Programming (WRS'07). ENTCS, vol. 204, pp. 71–82 (2008)
22. Palsberg, J., Jay, C.B.: The Essence of the Visitor Pattern. In: 22nd Intl. Computer Software and Applications Conf. (COMPSAC '98). pp. 9–15. IEEE Computer Society (1998)
23. Ren, D., Erwig, M.: A generic recursion toolbox for Haskell or: scrap your boilerplate systematically. In: Proceedings of the ACM SIGPLAN Workshop on Haskell. pp. 13–24. ACM (2006)
24. Reynolds, J.C.: The Discoveries of Continuations. Lisp and Symbolic Computation 6(3-4), 233–248 (1993)
25. Sloane, A.M.: Lightweight Language Processing in Kiama. In: Generative and Transformational Techniques in Software Engineering III, GTTSE 2009. Revised Papers. LNCS, vol. 6491, pp. 408–425. Springer (2011)
26. Tisi, M., Martínez, S., Jouault, F., Cabot, J.: Refining Models with Rule-based Model Transformations. Tech. Rep. 7582, INRIA (2011)
27. Veldhuizen, T.L.: Expression templates. C++ Report 7(5), 26–31 (Jun 1995), reprinted in C++ Gems, ed. Stanley Lippman
28. Viera, M., Swierstra, S.D., Lempsink, E.: Haskell, do you read me?: constructing and composing efficient top-down parsers at runtime. In: 1st ACM SIGPLAN Symposium on Haskell, (Haskell'08). pp. 63–74. ACM (2008)
29. Visser, E., Benaissa, Z., Tolmach, A.: Building program optimizers with rewriting strategies. In: ICFP'98: 3rd ACM SIGPLAN Intl. Conf. on Functional Programming. pp. 13–26. ACM Press (1998)
30. Visser, J.: Visitor combination and traversal control. In: OOPSLA '01: 16th ACM SIGPLAN Conf. on Object oriented programming, systems, languages, and applications. pp. 270–282. ACM (2001)