View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CWI's Institutional Repository

High-Fidelity Metaprogramming with Separator Syntax Trees

Rodin T. A. Aarssen
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Eindhoven University of Technology
Eindhoven, The Netherlands
Rodin.Aarssen@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
University of Groningen
Groningen, The Netherlands
T.van.der.Storm@cwi.nl

Abstract

Many metaprogramming tasks, such as refactorings, automated bug fixing, or large-scale software renovation, require high-fidelity source code transformations – transformations which preserve comments and layout as much as possible. Abstract syntax trees (ASTs) typically abstract from such details, and hence would require pretty printing, destroying the original program layout. Concrete syntax trees (CSTs) preserve all layout information, but transformation systems or parsers that support CSTs are rare and can be cumbersome to use.

In this paper we present separator syntax trees (SSTs), a lightweight syntax tree format, that sits between AST and CSTs, in terms of the amount of information they preserve. SSTs extend ASTs by recording textual layout information separating AST nodes. This information can be used to reconstruct the textual code after parsing, but can largely be ignored when implementing high-fidelity transformations.

We have implemented SSTs in Rascal, and show how it enables the concise definition of high-fidelity source code transformations using a simple refactoring for C++.

CCS Concepts • **Software and its engineering → Translator writing systems and compiler generators**; **Source code generation**; **Software maintenance tools**.

Keywords metaprogramming, high-fidelity code transformations

ACM Reference Format:
Rodin T. A. Aarssen and Tijs van der Storm. 2020. High-Fidelity Metaprogramming with Separator Syntax Trees. In *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '20), January 20, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3372884.3373162

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PEPM '20, January 20, 2020, New Orleans, LA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7096-7/20/01…$15.00
https://doi.org/10.1145/3372884.3373162

1 Introduction

Many metaprogramming tasks require *high-fidelity* source code transformations: transformations that preserve as much layout (comments, indentation, whitespace, etc.) of the input as possible. Typical examples include:

- Automated refactorings [12];
- Mass maintenance scenarios [14];
- Automated bug fixing [10].

High-fidelity metaprogramming further promotes end-user scripting of source code transformations, where programmers not only apply standard refactorings and restructurings offered by mainstream IDEs or dedicated tools (such as Coccinelle), but are able to script (one-off) transformations themselves, using knowledge about their code bases.

Unfortunately, high-fidelity is a high bar to reach. Most program transformation systems employ ASTs to represent source code. While valuable in the sense that they abstract from "non-essential detail", simplifying metaprogramming across the board, ASTs throw the baby out with the bath water from the perspective of high-fidelity transformation scenarios. Concrete Syntax Trees (CSTs), on the other hand, represent source code in exactly the way it was parsed, containing all syntactical information. While "unparsing" such trees gives back the original source code, CSTs usually reflect productions from some context-free grammar defining the source language, which may not be available.

In this paper we present Separator Syntax Trees (SSTs), a syntax tree representation that conveniently sits between the abstraction provided by ASTs, and the complexity of heavy-weight, full-blown CSTs (Section 2). We motivate SSTs using an example transformation, define their structure, and how they can be mapped back to source code. The use of SSTs is further illustrated using a simple term rewriting language (HIFITRS), detailing pattern matching, substitution, and how to deal with (separated) lists (Section 3). Finally, we discuss our implementation in the Rascal metaprogramming system [8], evaluating SSTs on the definition of a simple C++

RIGHTSLINK

refactoring (Section 4). Throughout the paper, we use Rascal in the meta program code examples.

1.1 Overview and Motivating Example

At the heart of many programming transformations, are rewrite rules such as the following:

```
ifThen(not(e), s1, s2) ⇒ ifThen(e, s2, s1)
```

This example uses pattern-matching against ASTs, matching conditional statements with a negated conditional. The rule replaces matching subtrees somewhere in the program with the pattern on the right-hand side where the then and else branches are swapped. Applying such rules to a program's AST produces another AST, which has to be rendered to actual source code again.

The example illustrates two problems for high-fidelity metaprogramming:

- Source code layout of the source is lost in translation;
- The new term constructed on the right-hand side has no layout at all.

As a result, such metaprograms require post-transformation pretty printing, inventing new layout for the transformed program. In many cases this is unacceptable, because it loses comments of the input source, and might make the transformed program look foreign to the developers of the code, because of coding conventions the pretty-printer is unaware of.

Systems such as ASF+SDF [3], TXL [4], and Rascal [8] support CSTs, which preserve all layout of the source program; additionally, such systems support concrete syntax patterns to allow rewrite rules to be expressed directly in the object language. For instance, the above rule could be expressed in Rascal as follows:

```
(Stmt)`if (!<Exp e>) <Stmt s1> else <Stmt s2>` ⇒
  (Stmt)`if (<Exp e>) <Stmt s2> else <Stmt s1>`
```

In this case the left-hand side matches against CSTs (ignoring layout), which means that `e`, `s1`, and `s2` will carry over all internal layout information, when substituted in the right-hand side. Conversely, the concrete pattern on the right-hand side defines the layout of the rewritten conditional, as part of the metaprogram.

In the previous example, the conditional on the right-hand side is written (and parsed) as a one-liner. The metaprogrammer could have specified a different layout, for instance[1]:

```
(Stmt)`if (!<Exp e>) <Stmt s1> else <Stmt s2>` ⇒
  (Stmt)`if (<Exp e>)
        '  <Stmt s2>
        'else
        '  <Stmt s1>`
```

[1]The single quotes are not part of the pattern, but allows concrete patterns to be indented nicely.

Concrete syntax trees solve a large part of the problem of high-fidelity metaprogramming, yet suffer from a number of drawbacks:

- Transformation systems that support CSTs are scarce, and even if CSTs are supported, adapting existing parsers to produce CSTs is non-trivial.
- Writing a grammar for a language from scratch in the formalism of systems that do support CSTs is a complex task [7], and hardly feasible for complex languages such as C++ and Cobol.
- Without support for concrete pattern matching, processing CSTs can be cumbersome.

Separator syntax trees represent a middle-ground between the fidelity of CSTs and the simplicity of ASTs. SSTs are like ASTs, except each node is annotated with a list of strings, representing the literal source code that separates the children of the node. SSTs support unparsing, like CSTs, to obtain the original textual source code.

In terms of metaprogramming, SSTs can be the basis of rule-based rewriting systems that support rules like the following:

```
ifThen(not(e), s1, s2) ⇒
  (Stmt)`if (<Exp e>) <Stmt s2> else <Stmt s1>`
```

In this case the left-hand side consists of an abstract pattern, which will be matched against an SST (modulo the separators). As a result, `e`, `s1`, and `s2` preserve their separator lists when inserted into the right-hand side. The right-hand side pattern is itself parsed into an SST, and hence will get the layout as specified in the rule itself.

SSTs are easy to construct, either during parsing, or afterwards if accurate source location information is available. This opens the way to high-fidelity metaprogramming in systems like Rascal or TXL, when parsing is realized by reusing external, black box parsers (cf. [1]).

2 Separator Syntax Trees

2.1 Introduction

As a compromise between ASTs and CSTs, we introduce Separator Syntax Trees (SSTs). SSTs contain the structure of abstract syntax, augmented with the separator strings that are present in source code, filling the gaps around and between abstract syntax nodes. Compared to CSTs, where such separator strings might come in different forms (e.g., whitespace, or a literal representing some operator), in SSTs this distinction is not made; the "non-essential" information in between AST nodes is recorded simply as text.

Listing 1 shows the meta-definition of SSTs in the form of the algebraic data type `Term`. The leaves of an SST are represented by (literal) `token` nodes, corresponding to tokens of a language. Their source representation is simply the argument `src`. Such nodes are used to encode identifiers, integer literals, and so on.

RIGHTSLINK

Listing 1. ADT definition of Separator Syntax Trees (SSTs).

```
data Term
 = token(str src)
 | cons(str name, list[Term] args, list[str] seps)
 | lst(list[Term] elts, list[str] seps);
```

The `cons` constructor represents an SST node, storing the SST constructor name in the `name` parameter, and the SST node's arguments in the `args` parameter. The final argument `seps` contains a list of strings, representing concrete syntax fragments that are part of the node's concrete syntax, but are not covered by the node's children.

The following invariants on SST nodes hold:

- $|seps| = |args| + 1$, because it contains the layout elements in between the arguments, as well as the layout before and after the span of the children;
- for nullary SST nodes (i.e., $|args| = 0$), the singleton string in `seps` is the whole source of the SST node.

Consider the following abstract grammar of a simple expression language:

```
data Expr
  = lit(str val)
  | paren(Expr e)
  | call(str name, list[Expr] args)
  | mul(Expr lhs, Expr rhs)
  | add(Expr lhs, Expr rhs);
```

If we now have a source text `(1)`, which corresponds to the AST `paren(lit("1"))` over the `Expr` data type, this is represented as the `Term` tree:

```
cons("paren",
  [cons("lit", [token("1")], ["", ""])],
  ["(", ")"])
```

Note how the outer parentheses are represented as the separators on the `paren` tree.

As another example, consider the AST node of `1+2`, which is `add(lit(1), lit(2))`. This AST node will correspond to the `Term`:

```
cons("add",
  [cons("lit", [token("1")], ["", ""]),
   cons("lit", [token("2")], ["", ""])],
  ["", "+", ""])
```

In this case, there is no layout before or after the AST node, so the `seps` argument of the outer `cons` starts and ends with the empty string, whereas the middle element corresponds to the source text of the plus operator.

The `lst` constructor represents lists, where the elements of the list are stored in the `elts` parameter, and the literal strings separating elements in the `seps` parameter. Lists correspond to the EBNF regular symbols like `S*`, `S+`, and `S?`. As such they don't have outer "separators". Hence the invariants on `lst` constructors are:

Listing 2. Yielding SSTs back to text.

```
str yield(cons(_, args, seps))
 = yieldL(args, seps);

str yield(lst(xs, seps, _))
  = yieldL(xs, ["", *seps, ""]);

str yield(token(x)) = x;

str yieldL(xs, list[str] seps)
  = ( seps[0] | it + yield(xs[i]) + seps[i+1]
      | int i ← [0..size(xs)] );
```

- if $|elts| > 0$ then $|seps| = |elts| - 1$;
- if $|elts| = 0$ then $|seps| = 0$.

Lists are used, for instance, in the AST definition of `call` nodes, which may have an arbitrary list of arguments. For instance, the expression `f(1, 2)` corresponding to AST node

```
call("f", [lit("1"), lit("2")])
```

is represented by the following `Term`:

```
cons("call",
  [token("f"),
   lst([cons("lit", [token("1")], ["", ""]),
        cons("lit", [token("2")], ["", ""])],
       [", "])],
   ["", "(", ")"])
```

Note in this case how the comma between the arguments is captured in the separators of the `lst` constructor, and how the parentheses of the function call are recorded in the separators of the outer `"call"` `cons` node.

2.2 Yielding SSTs back to text

The `Term` data type is a high-fidelity source code representation, containing literal layout information. When yielding source code from a `Term`, this literal layout information has to be inserted between the term's child nodes. This algorithm is given in Listing 2, which defines three cases for the `yield` function, each of which handles one of the `Term` variants. For constructors, the source code is yielded by starting with the first separator string, and then repeatedly alternatingly appending a (yielded) child node, and a separator. This is factored out into the auxiliary function `yieldL`[2].

Similarly, for lists, source code is yielded by alternating between list elements and separators, starting with a (yielded) list element. By adding an empty string before and after the `seps` list, we can reuse the `yieldL` function[3]. Finally, the source code of a literal (`token`) term is simply the string it wraps.

[2]The ternary reducer expression `( initial | reduce | generator )` resembles a fold function commonly found in functional languages, with **`it`** as a special variable containing the current reduct.

[3]The asterisk `*` is the splice operator: `[1,*[2,3]] == [1,2,3]`.

RIGHTSLINK

2.3 Creating Separator Syntax Trees

SSTs can easily be reconstructed from ASTs (or CSTs) if AST nodes are annotated with accurate source location information, in terms of (file) offsets and lengths, by reading out the respective location. For token-like nodes, the text representation follows from the token value. For a node $f(k_1, \ldots, k_n)$ starting at offset i with length l, if $n = 0$, the corresponding text is the source range $(i, i+l)$. If $n > 0$, the separators can be retrieved from the original source text from the consecutive text ranges:

$$(i, k_1.offset), \ldots, (k_j.offset + k_j.length, k_{j+1}.offset), \ldots, (k_n.offset + k_n.length, i + l)$$

where $j \in \{1, \ldots, n-1\}$. Alternatively, SSTs can be created during parsing, where such offset and length information is often available directly.

3 Rewriting with Separator Trees

3.1 Introduction

To illustrate how to rewrite source code using SSTs, we introduce a simple term rewriting model, in the form of a small language, HiFiTRS. This language allows the meta programmer to declare rewrite rules for a given object language, where the left-hand side is specified using an abstract syntax pattern (matching against SSTs), and the right-hand sides in concrete, textual syntax[4].

A simple example of a HiFiTRS rule is the following:

```
rule mul(lit("2"), x) ⇒ "<x> + <x>"
```

This rule matches against a `mul` constructor from some expression language and rewrites it to the string `<x> + <x>` if the multiplication consists of doubling an expression x. The left-hand side pattern matches against SSTs *modulo* separators. The right-hand side is parsed into an SST, after substituting the textual value of x, so that the layout of the metaprogram is used for the result of the substitution. The interpolated metavariables x are bound to the SST of the second operand of the multiplication, so retain their layout information.

3.2 Syntax

The abstract syntax of HiFiTRS itself is given in Listing 3. The left-hand sides are represented by the `Pattern` data type, which closely resembles the `Term` data type, but without the separator information (cf. Listing 1). Additionally, a rule's left-hand side may contain variables (`var`) to capture subtrees, and list variables (`lvar`) to capture list slices (sublists).

The right-hand side of a rule consists of concrete syntax in the object language, which is used to construct a replacement SST term on a successful match. This is modeled using the `Txt` data type: a string of `Elt` elements, which can be literal text (`txt`), interpolated simple and list variables (`var`, `lvar`), and marked text (`marked`). The `var` and `lvar` elements correspond to the bracketed interpolation (using < and >) in the concrete syntax of HiFiTRS.

Listing 3. Abstract syntax for HiFiTRS.

```
1  data Rule = rule(Pattern lhs, Txt rhs);
2
3  data Pattern
4    = cons(str name, list[Pattern] args)
5    | lst(list[Pattern] elts)
6    | token(str src)
7    | var(str name)
8    | lvar(str name)
9    ;
10
11 alias Txt = list[Elt];
12
13 data Elt
14   = txt(str src)
15   | marked(str src)
16   | var(str name)
17   | lvar(str name);
```

The `marked` element is needed to deal with separators in the result of a rewrite rule, which might have to be deleted if an interpolated sublist is empty. Consider, for instance, the following rule:

```
rule call("f", [args*]) ⇒ "f(<args*>, logger)"
```

This rule matches the arguments of a function call into a list variable *args*∗, and appends an extra argument to the call expression. Since the original call to `f` might not have arguments, *args*∗ might be empty, and substituting its text yield into the right-hand side will lead to the incorrect string `f(, logger)`.

The rules for separators in such lists are object language dependent, so there is no generic way to deal with this problem. Instead we require the metaprogrammer to indicate the separator that needs to be removed if a list variable will be empty using a special marker. In the case of the example, the correct way to write this rule in HiFiTRS would be:

```
rule call("f", [args*]) ⇒ "f(<args*>§, §logger)"
```

The text enclosed by the § signs corresponds to the `marked` constructor of the `Elt` data type. If *args*∗ is empty, the substitution algorithm of HiFiTRS will remove the `marked` node from the result, before parsing the textual result of this rule into an SST (cf. Section 3.5).

3.3 Semantics

Without loss of generality, we assume that the object language used with HiFiTRS rewrite rules is single-sorted. The

[4] Note that HiFiTRS gives an abstract view on SSTs, and that it is not intended to be used in practice. The Rascal implementation (cf. Section 4) allows arbitrary Rascal code to be executed at the right-hand side, as long as an appropriate SST is produced.

RIGHTSLINK

basic evaluation of a set of rules on the SST of the subject program is then as follows:

- Traverse the input SST in innermost fashion;
- when a rule R matches a subtree T with an environment *env* mapping left-hand side variables to bound subterms:
  - subsititute the yield of the bound variables in *env* in the right-hand side of R (with the substitution algorithm described below);
  - parse the resulting string into an SST and replace T with it;
- continue until no rules match anymore.

Note that at run time, the bound SSTs in the environment are first yielded and then inserted into the concrete syntax of the right-hand side, after which the reduct is parsed. The innermost traversal strategy ensures that newly inserted SSTs during the second step are targets for rewriting in future iterations.

3.4 Matching SSTs

The pattern matching algorithm for HIFITRS is shown in Listing 4. It supports non-linear matching (i.e., repeated variables in patterns are allowed), and implements matching modulo separators (this is covered by the auxiliary function `equalModSep`). The `match` function takes a Term, a Pattern, and an environment.

Matching a term to a variable adds a new binding to the environment, if it is not bound yet, otherwise it checks that the current term is equal to the bound term. Literal nodes are matched against literal patterns, if they have the same textual content. SST `cons` nodes match if they have the same constructor name and arity, and all arguments match. List nodes are matched using the auxiliary function `matchL`, shown in Listing 5. For all other cases, `match` throws a failure exception, which is used during list matching for local backtracking.

As an example, consider the rule from Section 3.1, of which the left-hand side is mapped onto the `Pattern` ADT as

```
cons("mul", [cons("lit", [token("2")]), var("x")])
```

and applying this rule to the `Term`

```
cons("mul",
  [cons("lit", [token("2")], ["", ""]),
   cons("lit", [token("3")], ["", ""])],
  ["", " * ", ""])
```

which corresponds to the expression `2 * 3`. The `match` algorithm starts off in the fourth alternative, as both the pattern and the term are `cons` nodes, and their names match (`"mul"`). As the pattern and term node have the same number of arguments, the algorithm recurses pairwise into the children. For the first child, this then boils to matching the pattern

```
cons("lit", [token("2")])
```

against the term

```
cons("lit", [token("2")], ["", ""])
```

Listing 4. Match algorithm for HIFITRS.

```
1  Env match(t, var(x), env) = env + (x: t)
2    when x notin env;
3
4  Env match(t, var(x), env) = env
5    when x in env, equalModSep(env[x], t);
6
7  Env match(token(x), token(x), env) = env;
8
9  Env match(cons(x, as, _), cons(x, bs), env)
10   = ( env | it + match(as[i], bs[i], it)
11           | i ← [0..size(as)] )
12   when size(as) == size(bs);
13
14 Env match(lst(xs, seps), lst(ys), env)
15   = matchL(xs, seps, ys, env);
16
17 default Env match(_, _, _) = { throw Fail(); };
```

Listing 5. List matching on SSTs.

```
1  Env matchL([], _, [], env) = env;
2
3  Env matchL([], _, [!lvar(_), *_], _)
4    = { throw Fail(); };
5
6  Env matchL(ts, seps, [lvar(x), *ps], env) {
7    for (i ← [0..size(ts)+1])
8      try {
9        sub = lst(ts[0..i], seps[0..i]);
10       if (x in env, !equalModSep(env[x], sub))
11         continue;
12       return matchL(ts[i..], seps[i..], ps,
13                     env + (x: sub));
14     }
15     catch Fail(): ;
16   throw Fail();
17 }
18
19 default Env matchL([t, *ts], seps, [p, *ps], env)
20   = matchL(ts, seps[1..], ps, match(t, p, env));
```

Again, the pattern and subject constructors share the same name, and their only child matches successfully through the third `match` alternative. Matching the second child of the `"mul"` `cons` nodes entails matching the pattern `var("x")` to the second `"lit"` `cons` node of the term. Since the variable `x` is not bound in the environment yet, matching succeeds through the first `match` alternative, binding `x` to the `3` literal. The matching algorithm now successfully terminates with an environment containing a mapping for the `x` variable.

The `matchL` function matches lists which may contain list variables. Since the length of such list variables is unknown in advance, the algorithm has to try out bindings to slices of

RIGHTSLINK

lists of increasing size, until the whole list matches, or fails. This way, the `matchL` function finds the first (shortest) match.

The base case is when both the subject and the pattern are the empty list, in which case the environment of bindings is returned, indicating success. If the subject list is empty, but there are still patterns to be matched which are not list variables, the match fails.

When `matchL` encounters a list variable (`lvar`), the algorithm tries successive slices of the subject list as binding for the list variable, and tries to match the rest of the pattern. If this fails, the next slice is tried, otherwise the algorithm returns successfully.

If both the head of the subject and the head of the pattern list are ordinary patterns, `matchL` recurses back to `match` and continues with the tail of both subject and pattern.

Consider the rule from Section 3.2 that adds a logger argument to function calls to some function `f`, applying it to the code fragment `f(1, /*higher*/2)` that maps to the SST

```
cons("call",
  [token("f"),
   lst([cons("lit", [token("1")], ["", ""]),
        cons("lit", [token("2")], ["", ""])],
       [", /*higher*/"])],
   ["", "(", ")"])
```

Note how the comment between the two arguments is stored next to the comma as the sole separator string in the `lst` constructor. Matching the simple pattern fragments happens as before; matching the list nodes is delegated to `matchL`, comparing the pattern `lst([lvar(args)])` to the `lst` constructor of the `"call"` `cons` node. Since the head of the pattern is a list variable, the `matchL` algorithm tries to match an increasing number of term elements, starting with zero elements. Matching an empty pattern list against a non-empty subject list subsequently fails, and `matchL` then backtracks locally and tries to bind the list variable `args` to a single element. Again, matching fails, and the algorithm backtracks. In the next iteration, both arguments of the subject list are bound to the `args` in the environment. Now, `matchL` continues to match an empty pattern list to an empty subject list, which succeeds through the first alternative. The list matching thus succeeds, with both arguments bound to the meta variable `args`. Note that the separator between the two arguments is propagated as well (cf. line 9).

3.5 Substitution algorithm for HifiTRS

The matching algorithm of HifiTRS tries to match the left-hand side of a rule by traversing the subject SST. On a successful match, this algorithm returns a variable environment, mapping the variables occurring in the left-hand side to the SST term they were bound to during matching.

Right-hand sides of rules are modeled as a series of elements, where each element is either a string literal, a simple or list variable, or a marked fragment (cf. Listing 3). The variable environment is used to replace variables occurring in the right-hand side with the SST terms bound to them.

Listing 6. Substitution algorithm for HifiTRS.

```
1  alias Env = map[str, Term];
2
3  str subst(txt, env)
4    = ( "" | it + e.src | e ← subst(txt, [], env) );
5
6  Txt subst([], hist, _) = hist;
7
8  Txt subst([token(x), *tail], hist, env)
9    = subst(tail, [*hist, token(x)], env);
10
11 Txt subst([var(x), *tail], hist, env)
12   = subst(tail, [*hist, token(yield(env[x]))],
13           env);
14
15 Txt subst([marked(x), *tail], hist, env)
16   = subst(tail, [*hist, marked(x)], env);
17
18 Txt subst([lvar(x), *tail], hist, env) {
19   lst = env[x];
20   if (lst.elts == []) {
21     if (hist != [], hist[-1] is marked)
22       hist = hist[..-1];
23     else if (tail != [], tail[0] is marked)
24       tail = tail[1..];
25   }
26   return subst(tail, [*hist, token(yield(lst))],
27                env);
28 }
```

The substitution algorithm is given in Listing 6. The substitution source code is constructed in a two-step process. In the first pass, starting at line 6, the algorithm walks through the list of elements, handling the head element and recursing to the tail, while propagating the intermediate result. Literal and marked elements are simply propagated (lines 8-9, 15-16).

Simple variables are handled by looking up the corresponding SST term in the environment, yielding it and appending it to the intermediate result by wrapping it in a literal element (lines 11-13). Similarly, for list variables, the corresponding list slice is retrieved from the environment, is yielded and appended to the intermediate result (`hist`).

Additionally, if the list variable is bound to the empty list, the algorithm checks whether the preceding (or, alternatively, following) element is a marked literal element. If this is the case, this element is removed (lines 18-28). Finally, if there are no elements left, the first pass concludes by returning the intermediate result (line 6).

After the first phase, the list of elements only contains literal and marked elements. The second phase consist of appending the string literals from these elements, yielding the

RIGHTSLINK

final replacement source code (line 4). Note that if marked elements were not removed due to an empty list interpolation, they become part of the output, as is needed.

Again, consider the last rule application from Section 3.4. The right-hand side is mapped onto the `Txt` ADT as

```
[txt("f("), lvar("args"),
 marked(", "), txt("logger)")]
```

Recall that in that example, pattern matching was successful and the `args` variable was bound to

```
lst([cons("lit", [token("1")], ["", ""]),
     cons("lit", [token("2")], ["", ""])],
    [", /*higher*/"])
```

The first part of the substitution algorithm traverses the list of textual elements. The first `txt` element remains untouched; the `lvar` is replaced by a `txt` fragment after yielding the list bound to `args`, which produces `"1, /*higher*/2"`. As `args` was not empty, no `marked` fragment is to be removed. The last two list elements are left in place, and the first substitution phase results in

```
[txt("f("), txt("1, /*higher*/2"),
 marked(", "), txt("logger)")]
```

The second substitution phase concatenates the strings values wrapped by the `txt` and `marked` constructors, yielding `f(1, /*higher*/2, logger)`.

Now, consider that the same rule is applied to the code fragment `f()`. The rule is still applied successfully, but the `args` variable is now bound to the empty list. During the first substitution phase, the algorithm now detects that `args` is empty (cf. line 20). The algorithm then checks surrounding elements for marked text, and removes the marked fragment trailing the `args` variable. The intermediate result of substitution then becomes

```
[txt("f("), txt(""), txt("logger)")]
```

which, after concatenation, finally yields `f(logger)`.

4 Evaluation

4.1 Introduction

In this section, we will discuss the implementation of SSTs in Rascal [8]. In Section 4.2 we will illustrate how this implementation allows rewriting similar to the rules of HiFiTRS, using an example to add a logger parameter as the final formal argument to function definitions. Finally, in Section 4.3, we will discuss our efforts of implementing SSTs for C++ in Rascal; we will show how to implement the Encapsulate Field refactoring using our SST implementation, and additionally show that we can write the left-hand side of a rewrite rule in concrete syntax as well, by employing the Concretely framework [1].

4.2 Implementation in Rascal

We have modified the Rascal pattern matching engine to support matching against SSTs, both using abstract patterns and concrete syntax patterns (using the Concretely framework [1]). Listing 7 shows an example transformation that adds a logger argument as the final formal parameter to function definitions, this time using abstract matching. In this code fragment, a syntax tree `sst` is traversed using the built-in **`visit`** statement, which tries to match the single case pattern at arbitrary depth of the tree.

Listing 7. Adding Logger as last formal argument.

```
1 visit (sst) {
2   case funDef(t, n, [*formals], [*body])
3   ⇒ parse(subst("<t> <n>(<formals*>§, §Logger l) {
4                 '  l.log(\"Entering <n>.\");
5                 '  <body*>
6                 '}"), #Decl)
7 }
```

In Rascal, SSTs for a particular language are represented using ADTs with *keyword parameters* to contain the seperator strings. Keyword parameters are optional parameters of constructors. The example presupposes an ADT with a constructor for `funDef`, for instance, like:

```
data Decl(list[str] seps = [])
  = funDef(Type typ, str name,
     SepList[Formal] formals, SepList[Stmt] body)
  | ...
  ;
```

In this definition all constructors of `Decl` get the extra list of separators, initialized with the empty list. Keyword parameters can be ignored during matching, as happens in Listing 7, where the left-hand side of the visit-case does not specify the `seps` parameter.

Note that the list of formal parameters and statements in the body use a special `SepList` data type, to allow lists of nodes to be annotated with seperators as well; this type is defined as follows:

```
data SepList[&T]
  = lst(list[&T] elts, list[str] seps);
```

Also, defining SSTs in this way makes it trivial to lower an SST to an AST. The optional `seps` keyword parameter can simply be dropped, and any occurrence of a `SepList` can be replaced by the list it wraps.

On a successful match, the left-hand side variables `t`, `n`, `formals`, and `body` are bound to actual SST subtrees. The right-hand side of the rule gives the new concrete syntax for the function definition: the function header is reconstructed with an extra `Logger` parameter, putting the function type, name and other parameters back in place. The function body is reconstructed by adding a call to the newly added logger, followed by the pre-existing function body. A call to the `subst` function correctly replaces the variable meta syntax for the variables by yielding the SST terms they were bound

RIGHTSLINK

to. Here, the `subst` function does not explicitly receive the variable environment (cf. Listing 6); the variable bindings are retrieved using reflection.

The interpolated variables are formatted exactly as they occurred in the original source code, using the literal SST information. For the list variables `formals` and `body`, this means that the original separators between the elements (e.g., a comma between parameters, newlines between statement) are put back in place again. The new function definition itself is formatted exactly the way it was written down at the right-hand side of the pattern.

Finally, the `parse` function calls the parser for the declaration type (`#Decl`), yielding the appropriate SST term to be inserted into the traversed SST.

4.3 Refactoring C++

We have implemented SSTs in Rascal for C++, for which the AST data types were already available. Instead of mapping these ADTs to the `Term` data type (cf. Listing 1), we have embedded the essential literal syntax information directly into the data type definitions.

ASTs that are produced by the parser are converted to SSTs by reading out the file the AST corresponds to, and filling the `seps` lists for every node and list, using their source location attributes (cf. Section 2.3)[5].

In this section, we evaluate SSTs by describing a simple C++ refactoring, based on SSTs. In object-oriented programming, encapsulation is the restriction of access to data members of classes. These members are not directly accessible from outside of the class, but the class provides public getter and setter methods to query and update members. Achieving encapsulation on a class requires two things. First, all members with public visibility must be made private[6]. Second, public getter and setter functions for such members must be added to the class. Member variables that are not public, as well as non-variable members should not be touched by the refactoring.

Previously, we have used abstract syntax to match on SSTs. While this allows for highly specific pattern specification, it requires the meta programmer to have in-depth knowledge of the intricacies of the abstract syntax. Concretely is a technique that allows specification of patterns in concrete syntax, converting these to abstract patterns under the hood [1]. We have adapted Concretely to produce SSTs, allowing us to write concrete syntax both in left-hand side and right-hand side patterns of rewrite rules. In contrast, Concretely as defined in [1] only supported concrete *matching*, but not high-fidelity *transformation*.

[5]Rascal's existing C++ parsing front-end unfolds preprocessor macros, which is necessary to be able to produce ASTs. Putting macros back in newly constructed source code is an orthogonal problem to rewriting with SSTs, and will not be addressed in this paper.

[6]In the refactoring described in this section, we chose to make public members private, and leave protected members as is.

Listing 8. Encapsulate field for C++ in Rascal using SSTs.

```
visit (sst) {
  case (Decl)`class <Name c> {
              '  <Decl* pre>
              '  public:
              '  <Decl* between>
              '  <Type t> <Name n>;
              '  <Decl* post>
              '};`
  : {
    if (hasPrivateOrProtected(between)) {
      fail;
    }
    name = capitalize(yield(n));
    src = subst(
      "class <c> {
      '  <pre*>§
      '  §public:
      '  <between*>§
      '  §private:
      '  <t> <n>;
      '  public:
      '  void set" + name + "(<t> val) {
      '    <n> = val;
      '  }
      '  <t> get" + name + "() {
      '    return <n>;
      '  }§
      '  §<post*>
      '}");
    insert parse(src, #Decl);
  }
}
```

Listing 8 shows how this refactoring can be implemented using SSTs. Again, an SST is traversed in a **`visit`** statement. The left-hand side of the case describes a pattern to match out class definitions with public fields, in concrete C++ syntax, interpolated with Rascal meta variables. When this pattern matches on a subtree, it is checked that the matched variable is indeed public, by ensuring no `private:` or `protected:` visibility labels occur between the public visibility declaration and the variable[7]. Then, the variable name is capitalized to allow the getter and setter functions to be in camel case. The meta variables are then substituted. Note that the list variables `pre`, `between`, and `post` have a marked source code fragments that will be removed if they are bound to empty lists, ensuring there are no unnecessary empty lines. Finally, this source code is parsed, and inserted into the SST with the **`insert`** statement.

Listing 9 shows an example C++ class definition on the left-hand side. This definition contains three members, of which only the declaration of `x` is a target for encapsulation.

[7]The **`fail`** keyword aborts the current match and makes the traversal algorithm backtrack.

RIGHTSLINK

Running the refactoring of Listing 8 yields the declaration that is given at the right-hand side. Indeed, the declarations for `foo` and `y` are left as is. A `private:` visibility label is inserted before the declaration of `x`, after which visibility is reset to public and the newly constructed getter and setter functions are inserted. Note that the comment between the public visibility label and the method declaration is preserved, as it was stores as part of the separator between these elements. The other comment, however, is lost, as this comment is stored in the separator list of the class body, but is not covered by either `between` or `t`. If required, one could explicitly access these separators through the class definition and, e.g., with a regular expression, try to match out any comments.

For comparison, Listing 10 shows the same refactoring, but implemented using CSTs. In terms of complexity and length, the SST-based and CST-based implementations are similar. In the CST-based implementation, there is no built-in support for conditionally including separators around list variables; possible errors would need to be taken care of explicitly. Furthermore, all interpolated meta variables are explicitly typed. The applicability check and name capitalization are placed in the **when** clause, followed by calls to the parser of the `Name` nonterminal to create names for the getter and setter function[8]. Compared to the SST-based approach, where the full pattern is parsed at runtime, each time the traversal algorithm finds an appropriate match, in the CST-based version the full concrete pattern is parsed once at compile-time; only the new CST nodes for the function names are parsed at run-time. Listings 8 and 10 show that the effort of implementing a refactoring in either formalism is similar; however, the SST formalism is much more lightweight and does not require a grammar in Rascal's grammar formalism.

5 Discussion and Related Work

Van den Brand and Vinju introduced rewriting with layout by adding explicit layout nodes in grammar productions [2]. The ASF+SDF interpreter preserves these layout nodes in a way similar to our SST implementation. The modifications to the interpreter proved to be insignificant. While we did not carry out benchmarks, this matches our observation for SSTs.

Kort and Lämmel discuss common concerns for rewriting systems, and propose annotated syntax trees as the preferred formalism for refactoring [9]. SSTs can be seen as a form of annotated syntax trees; in fact, in our implementation (cf. Section 4.3), we partly implemented SST features by annotating syntax trees. While this can be seen as a form of *code tangling*, such an addition to a data type is non-invasive

[8]In Rascal, only CSTs can be interpolated into a CST. The `[Type]source` construct calls the parser for the `Type` nonterminal with `source` as input.

Listing 9. Result of applying Encapsulate Field.

```
class C {
  public:
  //Important
  int foo();
  //target
  int x;
  protected:
  int y;
};
```

⇒

```
class C {
  public:
  //Important
  int foo();
  private:
  int x;
  public:
  void setX(int val) {
    x = val;
  }
  int getX() {
    return x;
  }
  protected:
  int y;
};
```

Listing 10. Encapsulate field for C++ in Rascal using CSTs.

```
1  visit (sst) {
2    case (Decl)`class <Name c> {
3               '  <Decl* pre>
4               '  public:
5               '  <Decl* between>
6               '  <Type t> <Name n>;
7               '  <Decl* post>
8               '};`
9    ⇒ (Decl)`class <Name c> {
10            '  <Decl* pre>
11            '  public:
12            '  <Decl* between>
13            '  private:
14            '  <Type t> <Name n>;
15            '  public:
16            '  void <Name setter>(<Type t> val) {
17            '    <Name n> = val;
18            '  }
19            '  <Type t> <Name getter>() {
20            '    return <Name n>;
21            '  }
22            '  <Decl* post>
23            '};`
24     when !hasPrivateOrProtected(between),
25       str name := capitalize("<n>"),
26       Name setter := [Name]"get<name>",
27       Name getter := [Name]"set<name>"
28 }
```

– for example, pattern matching on syntax trees was not influenced by the extension.

The second issue, *persistent normalisations*, is not applicable to SSTs, as no simplification or normalization occurs in the process of creating SSTs. Similarly, we do not insert

RIGHTSLINK

low-level annotations into the object code or tree. If a preprocessing phase is necessary to gather information, it is possible to do this offline.

Restriction to tree-shaped data is discussed as the final concern, arguing that with tree-shaped data, it is impossible to simply navigate between two non-related nodes (e.g., navigating from call site to declaration site). We see this as an issue of the underlying meta programming environment, and not as a problem of the data structure. In our SST implementation, it is straightforward to generate relations over SST nodes by tree traversal.

The Haskell Refactorer, HaRe, is a fully functional tool for high-fidelity transformations on Haskell code [11]. Origin information is maintained both in ASTs and in the token stream, allowing locations to link AST nodes and tokens. Transformations are carried out both on AST and token stream simultaneously. Finally, refactored source code is produced from the token stream, rather than from the AST. Similarly to SST-based refactorings, refactoring targets are found using matching on syntax trees. As the name suggests, however, HaRe specifically targets Haskell as the object language.

Hills et al. have described their efforts in successfully implementing a script-supported refactoring in Rascal of the Rascal interpreter, in which the Visitor pattern was changed to the Interpreter pattern (V2I) [5]. In this context, a *refactoring script* is a meta program that may construct, analyze and transform models of software. We see SSTs as an interesting implementation vehicle for such *ad hoc* refactorings. The V2I refactoring employs string templates to generate source code, leaving it vulnerable to generating type-incorrect source code. Since source code is parsed during SST construction, SST-based transformations provide syntax-safety.

Coccinelle is a matching and transformation tool for C code, aimed at collateral evolution and bug fixing [10]. It comes with the SmPL language in which *semantic patches* are specified declaratively. A semantic patch differs from a regular patch in that it abstracts away from irrelevant details such as whitespace and variable names. As such, a semantic patch can modify any number of files. Just like with SSTs, in SmPL, concrete C syntax is used, intertwined with meta variables. Contrary to SSTs, semantic patches are matched against the Control Flow Graph of a piece of code.

De Jonge and Visser have developed an algorithm to preserve layout for Spoofax-based code transformations [6]. Arguing that simply pretty printing an AST is not desirable, the algorithm yields incremental text patches, based on origin tracking, and non-affected parts of the source code are left untouched. The algorithm includes heuristics whitespace adjustment around newly constructed terms, and for comment migration, based on work of Van de Vanter [13]. Newly constructed AST nodes are pretty printed, contrary to SST-based transformations, where the concrete syntax of the meta program is used. Since the algorithm requires an underlying grammar, the distinction between layout and comments can be made, allowing better control over comments than with SSTs.

Coining the *high-fidelity* term in the scope of code transformations, Waddington and Yao have discussed their transformation system Proteus, supporting high-fidelity C++ code transformations [15]. As the underlying formalism, Proteus uses Literal-Layout ASTs (LL-ASTs), which augment simple ASTs first by adding nodes representing literals from their own grammar productions, and second by introducing layout nodes between every node.

Transformations are specified in the YATL language, which uses Stratego primitives under the hood. Specifically aimed at refactoring C++ code, macro expansions and includes are tracked by inserting low-level annotations (cf. [9]). Construction of LL-ASTs for Expressions and Statements is possible using concrete C++ syntax.

In SSTs, there is no distinction between literal information stemming from production literals, or layout literals. The creation of LL-ASTs requires a concrete grammar to decide which tokens are literal or layout nodes, whereas SSTs do not require a grammar to reconstruct such information. Our SST implementation for C++ also allows for the construction of SST nodes using concrete syntax, and supports more nonterminals to be parsed besides Statements and Expressions. Proteus has some heuristics about inserting layout and removing comments. In the context of SSTs, it is not possible to distinguish between layout and comments.

5.1 Limitations

The precedence of operators is not modeled in the SST formalism. For example, in the Expr language described in Section 2.1, it is possible to write rewrite terms into terms that have wrong precedence: consider, for instance, rewriting a literal `2` to `1+1` on a code fragment `2*3`, which would incorrectly yield `1+1*3`. Using parentheses, the meta programmer can force correct binding; however, always introducing parentheses may yield redundant parentheses in the result. There is no general solution to this. However, if the precedence relation is known – and if parentheses are part of the abstract syntax, as is the case in Expr – extra rewrite rules could be added that, based on the precedence relation, remove redundant nesting.

If code fragments are moved, they might end up in a position with a different desired indentation depth than at the original position. In order for the target code to look well-formatted, the indentation level would need to be adjusted for the inserted code. Again, there is no general solution to this problem, but it is possible to define language-specific heuristics, which could be injected into the `subst` function (cf. Section 3.5) to fix indentation.

RIGHTSLINK

6 Conclusion

Many source code transformations, such as automated refactorings, require high-fidelity: transformations that preserve as much of the textual layout and comments of the original code as possible. Concrete Syntax Trees (CSTs) are a solution to this problem, but they are heavy-weight, not available in many transformation systems, and not easy to construct from pre-existing parsers. In this paper we have introduced Separator Syntax Trees (SSTs), a simple middle-ground between ASTs and CSTs, which can be used to implement high-fidelity metaprograms.

SSTs maintain textual layout between AST nodes as lists of strings. This allows accurate reconstruction of the original source code from them. We defined the meta-datatype of SSTs, and showed how they enable high-fidelity rewriting using a simple term rewriting language HIFITRS. In particular, we showed how to deal with pattern matching modulo separators, and substitution in the presence of potentially empty list variables.

SSTs have been implemented in the Rascal [8] metaprogramming system, as an extension of the CONCRETELY framework [1]. This supports high-fidelity source code rewriting using Rascal's built-in pattern matching and rewriting engine, with concrete syntax patterns. We show the viability of the approach with a simple refactoring on C++, to encapsulate fields.

We hypothesize that SSTs provide a convenient syntax tree format, combining the light-weight simplicity of ASTs with the fidelity and syntax safety of full-blown CSTs.

Future work includes optimizing the implementation to avoid repeated parsing of concrete patterns, and applying SSTs in the context of large-scale transformation on realistic languages.

Acknowledgments

The authors would like to thank Jurgen Vinju and the anonymous referees for their valuable suggestions. The first author was supported by NWO under grant BISO.10.04: "Model Extraction for Re-engIneering Traditional Software (MERITS)", in collaboration with Philips Healthcare, Best, The Netherlands.

References

[1] Rodin T. A. Aarssen, Jurgen J. Vinju, and Tijs van der Storm. 2019. Concrete Syntax with Black Box Parsers. *The Art, Science, and Engineering of Programming* 3, 3 (2019). https://doi.org/10.22152/programming-journal.org/2019/3/15

[2] Mark G. J. van den Brand and Jurgen J. Vinju. 2000. Rewriting with layout. In *Proceedings of the First International Workshop on Rule-Based Programming (RULE'00)*.

[3] Mark van den Brand, Arie van Deursen, Jan Heering, Hielkje de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. 2001. The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. *Electronic Notes in Theoretical Computer Science* 44, 1 (2001), 3–8. https://doi.org/10.1016/S1571-0661(04)80917-4

[4] James R. Cordy. 2004. TXL – A Language for Programming Language Tools and Applications. *Electronic Notes in Theoretical Computer Science* 110 (2004), 3–31. https://doi.org/10.1016/j.entcs.2004.11.006

[5] Mark Hills, Paul Klint, and Jurgen J. Vinju. 2012. Scripting a Refactoring with Rascal and Eclipse. In *Proceedings of the Fifth Workshop on Refactoring Tools (WRT'12)*. ACM, New York, NY, USA, 40–49. https://doi.org/10.1145/2328876.2328882

[6] Maartje de Jonge and Eelco Visser. 2012. An Algorithm for Layout Preservation in Refactoring Transformations, In Software Language Engineering. *Lecture Notes in Computer Science* 6940, 40–59. https://doi.org/10.1007/978-3-642-28830-2_3

[7] Paul Klint, Ralf Lämmel, and Chris Verhoef. 2005. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering and Methodology* 14, 3 (2005), 331–380. https://doi.org/10.1145/1072997.1073000

[8] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'09)*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[9] Jan Kort and Ralf Lämmel. 2003. Parse-tree annotations meet re-engineering concerns. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*. 161–170. https://doi.org/10.1109/SCAM.2003.1238042

[10] Julia L. Lawall, Julien Brunel, Nicolas Palix, René R. Hansen, Henrik Stuart, and Gilles Muller. 2009. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'09)*. IEEE, 43–52. https://doi.org/10.1109/DSN.2009.5270354

[11] Huiqing Li, Simon Thompson, and Claus Reinke. 2005. The Haskell Refactorer, HaRe, and its API, In Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications. *Electronic Notes in Theoretical Computer Science* 141, 4, 29–34. https://doi.org/10.1016/j.entcs.2005.02.053

[12] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, 552–576. https://doi.org/10.1007/978-3-642-39038-8_23

[13] Michael L. Van De Vanter. 2001. Preserving the documentary structure of source code in language-based transformation tools. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'01)*. 131–141. https://doi.org/10.1109/SCAM.2001.972674

[14] Niels P. Veerman. 2007. Automated Mass Maintenance of Software Assets. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. 353–356. https://doi.org/10.1109/CSMR.2007.15

[15] Daniel G. Waddington and Bin Yao. 2005. High-Fidelity C/C++ Code Transformation, In Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications. *Electronic Notes in Theoretical Computer Science* 141, 4, 35 – 56. https://doi.org/10.1016/j.entcs.2005.04.037

RIGHTSLINK