Scrap Your Reprinter

A Datatype Generic Algorithm for Layout-Preserving Refactoring

Harry Clarke
School of Computing
University of Kent
hc306@kent.ac.uk

Vilem-Benjamin Liepelt
School of Computing
University of Kent
vl81@kent.ac.uk

Dominic Orchard
School of Computing
University of Kent
d.a.orchard@kent.ac.uk

ABSTRACT

Refactoring tools are extremely useful in software development: they provide automatic source code transformation tools for a variety of laborious tasks like renaming, formatting, standardisation, modernisation, and modularisation. A refactoring tool transforms part of a code base and leaves everything else untouched, including secondary notation like whitespace and comments. We describe a novel datatype generic algorithm for the last step of a refactoring tool – the *reprinter* – which takes a syntax tree, the original source file, and produces an updated source file which preserves secondary notation in the untransformed parts of the code. As a result of being datatype generic, the algorithm does not need changing even if the underlying syntax tree definition changes. We impose only modest preconditions on this underlying data type. The algorithm builds on existing work on datatype generic programming and zippers. This is useful technology for constructing refactoring tools as well as IDEs, interactive theorem provers, as well as verification and specification tools.

1 INTRODUCTION

Refactoring tools act a bit like compilers: they read in source code and convert it into a machine representation upon which some transformations are performed. However, they differ from compilers in that they output textual source code in the language that was originally input. To be of any use, they must preserve the lexical style of all untransformed code, such as comments and whitespace (*secondary notation* or *documentary structure* [13]). Compilers on the other hand discard any parts of the source text that are not program code. We refer to generation of the output source code in a refactoring tool as the *reprinter*–the last step where an AST is converted back into source text:

$$\begin{tikzcd} \textit{source text} \arrow[r, "\textbf{lex}"] & \textit{tokens} \arrow[r, "\textbf{parse}"] & \textit{AST} \arrow[loop above, "\textbf{transform}"] \arrow[r, "\textbf{reprint}"] & \textit{source text} \end{tikzcd}$$

As a simple running example, consider an SSA-like language with variables, integer addition in prefix notation, and constants. The following is an example program in our source language:

```
x = +(1,2)
y  =  +(x, 0)
// Calculate z
z  =  +( 1,  +(+(0,x) ,y) )
```

Note that for demonstration purposes this example deliberately uses varying amount of white space around the assignments and operations.

Imagine a refactoring that removes redundant additions of 0, *i.e.* performs rewrites +(e, 0) $\rightsquigarrow e$ and +(0, e) $\rightsquigarrow e$ for some expression e. The refactoring and reprinting should produce the following output source text, preserving whitespace and comments:

```
x = +(1,2)
y  =  x
// Calculate z
z  =  +( 1,  +(x ,y) )
```

An implementation cannot simply pretty print the transformed AST as this would destroy the secondary notation.

One solution is to store as much of the secondary notation as possible in the AST (either via specialised nodes or annotations) to provide a layout-preserving pretty printing (see, *e.g.* [6, 7, 14]). However, this requires a large engineering effort, does not integrate well with many front-end generation tools (for example, most lexers readily throw away additional whitespace), and is extremely difficult when multi-line comments are mixed with multi-line syntactic elements (see discussion by de Jonge *et al.* [3]). Another solution uses *text patching*, where AST changes are converted to edit instructions on the source code (*e.g.*, *diffs*), coupled with heuristics about layout adjustments [3].

We propose a new, simple approach that is language agnostic. Our reprinting algorithm combines an updated AST with the original source text to produce an updated source file. We detail a datatype generic implementation which can be applied to any algebraic data type satisfying a minimal interface for providing source code locations. Genericity means the algorithm does not need reimplementing even if the underlying AST datatypes are changed or extended. Furthermore, our reprinting algorithm has the advantage that an implementer need not write a pretty printer for all parts of their syntax tree, only for those parts which might get refactored, requiring fresh source text generation.

Our implementation is based on the datatype generic facilities provided in GHC (the Glasgow Haskell Compiler) [8, 9] and a datatype generic zipper construct [1] for simplifying the datatype generic traversal.

Reprinting is useful not just in refactoring tools but also in IDEs (for example, with live macro expansions), interactive theorem provers, or specification systems where a tool might automatically generate specifications into an existing code base. Our algorithm

IFL'17, 30th August-1st September 2017, Bristol, UK

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

is used in the CamFort tool which provides refactoring of Fortran code [11] and several verification features which use the reprinting algorithm to insert inferred specifications into source code [2].

Roadmap. We start with an informal overview of the algorithm (Section 1.1). Section 2 provides some background on zippers and datatype generic programming. Section 3 outlines the algorithm in detail, including its GHC/Haskell implementation. Section 4 shows that reprinting and parsing form a bidirectional lens, which gives a framework for reasoning about reprinting.

Section 5 generalises the algorithm, weakening its preconditions. We present this separately to the first algorithm since the first algorithm is conceptually simpler yet still catches a wide range of situations. Section 6 concludes with some discussion of the implications of our pre-conditions and some further work.

Our code is available as a package (see http://hackage.haskell.org/package/reprinter), which includes the examples used here.

1.1 Illustrated example

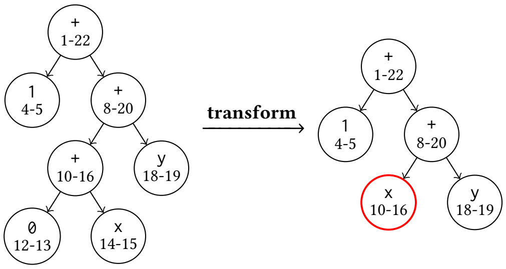
Consider the expression from the last line of the preceding example, which is refactored by removing redundant additions of 0:

$$\texttt{+(~1,~~~+(+(0,x)~,y)~)} \xrightarrow{\textbf{transform \& reprint}} \texttt{+(~1,~~~+(x~,y)~)}$$

To make the whitespace preservation here clear, the following shows the column number for the source text before and after transformation and reprinting:

col #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
pre	+	(		1	,			+	(	+	(	0	,	x	)		,	y	)		)
post	+	(		1	,			+	(	x		,	y	)		)					

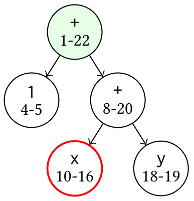
In terms of the abstract syntax trees, the refactoring is represented as follows, where each AST node is annotated with the *source code span*: a pair of column numbers marking the extent of the AST node's origin in the original source text.



The refactored node on the right is lined in red and notably has the original source span preserved. For the full algorithm, this source code span also includes the line numbers, but for simplicity we elide this here since the example is located on a single line.

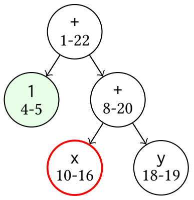
The algorithm performs a depth-first traversal of the AST and applies a pretty printing on any refactored nodes. In the actual implementation (Section 3), the reprinting algorithm is parameterised by a generic function which we call a *reprinting*, which may provide pretty printing for various different types of node.

In the illustration here, we mark the currently visited node in light green. The depth-first traversal of the AST simultaneously "traverses" the input text, which is statefully consumed. At each step we record the state, which comprises the remaining input text and a source code position (called the *cursor*) which signifies our position in the original source text. We also show the partially-built output source code, which grows as we traverse the tree.



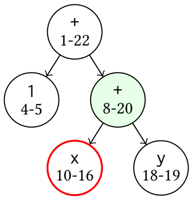
cursor = 1
input = `"+( 1,   +(+(0,x) ,y) )"`
output = `""`

Since the root node is not refactored, the algorithm proceeds to the first child (actions we refer to as **down** and **enter**):



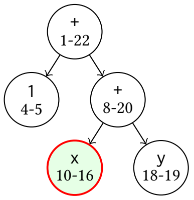
cursor = 1
input = `"+( 1,   +(+(0,x) ,y) )"`
output = `""`

Since the current node is not refactored and it has no children, we proceed to the next sibling (**right** and **enter** actions):



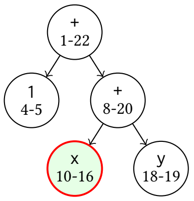
cursor = 1
input = `"+( 1,   +(+(0,x) ,y) )"`
output = `""`

Since this node is not refactored, we proceed to its first child (**down** and **enter** again):



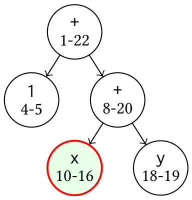
cursor = 1
input = `"+( 1,   +(+(0,x) ,y) )"`
output = `""`

Since this node is marked as refactored, we take the substring of the input source code from the cursor to the lower-bound position of this node (column 10) and add it to the output source code, consuming this part of the input source. The cursor is also updated to the lower bound of the node:



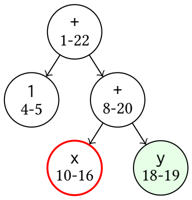
cursor = 10
input = `"+(0,x) ,y) )"`
output = `"+( 1,   +("`

Next, since the node is being refactored, we apply a pretty printing algorithm to generate a fresh piece of source text for this node which is then added to the output text. We also delete the portion of the input source text between the lower and upper bounds of the node, and update the cursor to the upper bound (column 16):



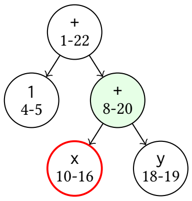
```
cursor = 16
 input = " ,y) )"
output = "+( 1,  +(x"
```

This node is now processed so we move to the next right sibling:



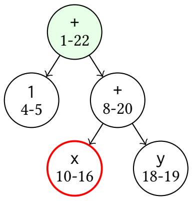
```
cursor = 16
 input = " ,y) )"
output = "+( 1,  +(x"
```

This node is not refactored and it has no children or siblings so we return to the parent node:



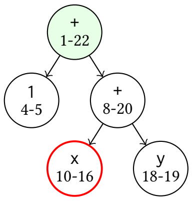
```
cursor = 16
 input = " ,y) )"
output = "+( 1,  +(x"
```

This node has no siblings so we return to the parent node:



```
cursor = 16
 input = " ,y) )"
output = "+( 1,  +(x"
```

This node has no siblings and no parent. On returning to the root, we append to the output text the remaining input source text:



```
cursor = 22
 input = ""
output = "+( 1, +(x ,y) )"
```

This concludes the algorithm and we now have the reprinted output, preserving all the whitespace in the non-transformed syntax.

Section 3 gives the implementation of this algorithm in detail. First we provide background on "zippers", the underlying data structure used to implement the tree traversal described above.

2 BACKGROUND

A zipper is an alternate representation of a datatype that allows subparts of a piece of data to be focussed upon, while preserving the rest of the datatype [5]. The *focus* can then be shifted around the datatype, maintaining the rest of the data structure as the *context*. The focus is shifted by navigation operations. Our algorithm is defined in terms of a generic zipper construction [1] to provide datatype generic traversal.

2.1 Standard Zippers

Zippers facilitate traversing and transforming a data structure and provide constant-time updates.

A zipper is made up of two parts:

- Focus - the part of the data structure we are currently viewing;
- Context - the rest which is "out-of-focus".

A zipper for lists can be defined like so:

```
type ListZipper a = ([a], ListContext a)
type ListContext a = [a]
toListZipper :: [a] → ListZipper a
toListZipper l = (l, [ ])
```

When we first create a zipper, the focus represents the entire data structure (in our case, the entire list), as we are at the root node, and the context is the empty list. Note that because lists are one-dimensional, we can represent the context also as a list; however, as we will see in our next example, not every data structure exhibits this symmetry.

Navigation operations shift the focus of the zipper. For the list zipper we have:

```
listDown :: ListZipper a → Maybe (ListZipper a)
listDown (x : xs, ys) = Just (xs, x : ys)
listDown ([ ], ys)     = Nothing

listUp :: ListZipper a → Maybe (ListZipper a)
listUp (xs, y : ys) = Just (y : xs, ys)
listUp (xs, [ ]) = Nothing
```

Thus, moving "down" a list navigates into the focus, extending the context with the element from the top of the focus. If the focus is empty, then we return Nothing, *i.e.*, there are no children left to move down into. Moving up unfolds the context, extending the focus. Once the empty context is reached, we are at the top/beginning of the list, and there is nowhere left for us to go.

Thus, for *toListZipper* $[1, 2, 3]$, we get the following navigation (eliding the Just constructor of the Maybe type in the results):

$$\begin{tikzcd}
([1,2,3],[\,]) \arrow[d, bend left, "listDown"] \\
([2,3],[1]) \arrow[u, bend left, "listUp"] \arrow[d, bend left, "listDown"] \\
([3],[2,1]) \arrow[u, bend left, "listUp"] \arrow[d, bend left, "listDown"] \\
([\,],[3,2,1]) \arrow[u, bend left, "listUp"]
\end{tikzcd}$$

Zippers on trees have a similar structure with the added complication that the context cannot have the same shape as the focus. Huet [5] shows how a tree can be converted into a tree zipper, where the context contains the parent, and the siblings on the left and right of the focus:

```
data Tree a = Item a | Section [Tree a]
type TreeZipper a = (Tree a, TreeContext a)
data TreeContext a
    = TRoot | TNode (TreeContext a) [Tree a] [Tree a]
```

Navigation operations for the tree zipper allow the focus to be moved *up* (to a parent) or *down* (to child nodes), or *left* and *right* (between siblings). We refer the reader to Huet's work for the definitions of the navigation operations for the tree zipper [5].

Fortunately, a zipper representation of a *regular* datatype can be calculated from its algebraic representation via Leibniz differentiation [10], thus enabling generic zippers for many common data types. This technique is leveraged to generate zippers automatically.

2.2 Generic zippers

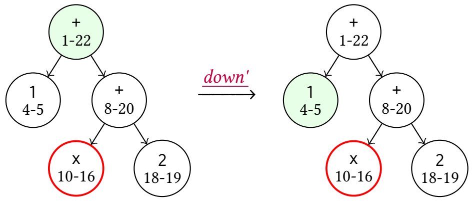
Scrap Your Zipper (SYZ) [1] is a Haskell library that allows the programmer to traverse any tree structure with ease by automatically converting a datatype into its zipper representation. The library provides various operations for navigation, insertion, and transformation of zippers. The only requirement is that all parts of the datatype have an instance of Scrap Your Boilerplate's [8] *Data* class, providing datatype generic functionality. This makes it an excellent starting point for generic AST traversal and, subsequently, reprinting.

We highlight functions from the SYZ library by formatting them like *this* to make clear where they are used in the algorithm.

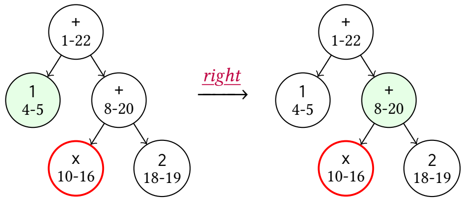
A function *toZipper* :: Data *a* ⇒ *a* → Zipper *a* maps any Data type into the zipper representation, enabling us to use the zipper navigation operations provided by SYZ.

The *down'* function of type Zipper *a* → Maybe (Zipper *a*) moves the focus of the zipper to the leftmost child of the current node (whereas the function *down*, which we do not use here, moves to the rightmost child).

For example, in our illustration the light green highlighted node corresponds to the focus node and *down'* has the following example behaviour (returning a Just value) when the focus starts at the root:



The other zipper navigation function we use is *right*, also of type Zipper *a* → Maybe (Zipper *a*), which moves the focus of the zipper to the right sibling. For example, in terms of our illustration:



The only other additional part of SYZ we use is the application of a generic query to the current focus, provided by the *query* functions:

```
-- Type of a generic query, from Data.Generics.Aliases
-- of Scrap Your Boilerplate
type GenericQ b = ∀c.Data c ⇒ c → b
```

```
-- Applies a generic function to a zipper and outputs the result.
query :: GenericQ b → Zipper a → b
```

Thus a generic function, defined for all types *c* satisfying Data producing a value of type *b*, is applied to the current focus of the zipper to produce a *b* value. Concretely, in our case, we will query an AST node for its (potential) refactored output source text.

3 DATATYPE GENERIC REPRINTING; IMPLEMENTATION

The reprinting algorithm combines an AST of the refactored program and the input source text into an output source text by a simultaneous traversal of both the AST and source text. The AST is traversed in depth-first order and the input source is traversed linearly. This relies on the AST containing information about the origin of its nodes in the original source file, along with some well-formedness conditions of this location information:

Definition 1. An AST is *source coherent* if every refactorable node has a *source span* associated with it, that is a pair of source code positions (*e.g.* line and column number) marking the lower bound and upper bound extent of the node's origin within the original source code. Furthermore, the AST must satisfy the following properties:

(1) *Enclosure*: the lower-bound position of a parent is less than or equal to the lower-bound of the first child, and the upper-bound position of a parent is greater than or equal to the upper-bound of the last child.
(2) *Sequentiality*: the upper-bound position of a node with a right sibling is less than or equal to the lower-bound position of its sibling.

Corollary 1. A depth-first traversal of a source coherent AST visits nodes in *source-span order*, *i.e.*, each visited node has a span which is greater than or equal to the previously visited node's span.

The two conditions of source coherence hold for the example here, which is defined in full below.

Note that usually an AST definition spans many mutually recursive datatypes (*e.g.*, a data type of blocks, statements, expressions, and so on). Therefore, source coherency must apply to all data types which represent refactorable parts of the overall AST.

We implicitly restrict ourselves to tree data structures, *i.e.*, data structures with no cycles. The *enclosure* property above goes some way to ruling out cycles. However, there may be cycles of nodes with the same span, which is not ruled out by enclosure. Therefore, an acyclic AST is an additional precondition to our algorithm.

We split our description of the algorithm into three parts: the main types (Section 3.1), the core of the algorithm provided by the top-level function *reprint* and the intermediate functions *enter* and *splice* (Section 3.2) and finally helpers for building "reprintings" (generic functions parameterising *reprint*) (Section 3.3). Section 3.4 provides two worked examples.

Section 5 gives a slightly more general algorithm which allows the sequentiality requirement of source coherency to be relaxed.

3.1 Types

Source text is represented by Text from the module Data.Text.Lazy which provides efficient operations and supports Unicode. We create an alias Source for convenience. The monoidal append operator <> is used for concatenation and *mempty* for empty source text.

```
import qualified Data.Text.Lazy as Text
import Data.Monoid ((<>), mempty)

type Source = Text.Text
```

The algorithm is not conceptually dependent on this particular representation of input source text. Other representations of the source text can be used as long as they are monoidal.

Positions in source code are given by pairs of line number and column number, which are 1-indexed and wrapped in newtype constructors to provide type safety:

```
newtype Line = Line Int deriving (Data, Eq, Ord, Show)
newtype Col = Col Int deriving (Data, Eq, Ord, Show)
type Position = (Line, Col)
initPosition = (Line 1, Col 1)
```

The span of a node of source text can be represented by a pair of positions, giving the lower and upper bound positions of the node.

```
type Span = (Position, Position)
```

The top-level function *reprint* has type:

```
reprint :: (Monad m, Data ast)
          ⇒ Reprinting m → ast → Source → m Source
```

which is a higher-order function parameterised by a "reprinting" function: a generic operation for replacing refactored nodes with fresh output text (*e.g.*, by pretty printing). Reprintings have type:

```
type Reprinting m = ∀node.Typeable node ⇒
   node → m (Maybe (RefactorType, Source, Span))
data RefactorType = Before | After | Replace
```

Thus, a reprinting maps any Typeable type *node* to Maybe of a triple of the refactoring type, the new source text for this node, and its span. We provide some helper functions for constructing a Reprinting in Section 3.3.

Note that *reprint* and Reprinting are parameterised by some monad *m*. This provides extra power if an application wishes to include some side effects as part of the reprint algorithm. For example, additional state can be useful when pretty printing refactored nodes, such as the number of lines deleted or added in order to balance newlines. We show an example of this in Section 3.4.

3.2 Zipper traversal; core algorithm

The zipper traversal part of the algorithm is performed by the function *enter*, of type:

```
enter :: Monad m ⇒ Reprinting m → Zipper ast
    → StateT (Position, Source) m Source
```

where Zipper is the zipper for some type *ast*, provided by the Scrap Your Zipper library [1]. The implementation is shown in Figure 1. Recall that the zipper operations are highlighted like *this*.

```
enter :: Monad m ⇒ Reprinting m → Zipper ast
    → StateT (Position, Source) m Source
enter reprinting zipper = do
    -- Step 1: Apply a refactoring
    refactoringInfo ← lift (query reprinting zipper)
    -- Step 2: Deal with refactored code or go to children
    output ← case refactoringInfo of
      -- No refactoring; go to children
      Nothing → go down'
      -- A refactoring was applied; splice the text.
      Just r → splice r
    -- Step 3: Enter the right sibling of the current context
    outputSib ← go right
    -- Append output of current context/children and siblings
    return (output <> outputSib)
  where
    -- Navigate in a particular direction.
    go direction =
      case direction zipper of
        -- Recursively apply 'enter' to the new focus
        Just zipper → enter reprinting zipper
        -- Otherwise return the empty source
        Nothing → return mempty
```

Figure 1: Core traversal of the reprinting algorithm

The *enter* function provides the main functionality. It firstly applies the generic *reprinting* function of type Reprinting *m* to the current node, yielding information about whether a reprinting was performed or not. Recall that the *reprinting* function returns a value of type *m* (Maybe...). Now *lift* in Step 1 raises the monadic computation in *m* to StateT *s m*, so *refactoringInfo* has type Maybe (RefactorType, Source, Span).

The second step matches on the information returned from the reprinting to determine whether to perform some splicing of the input source text or to navigate to the children. If *refactoringInfo* is Nothing (implying no refactoring) then *down'* is called via the *go* helper function which proceeds to the children if they exist, else returning the empty source text. Otherwise, we have Just *r* where *r* is a triple of the refactoring type (RefactorType), new output source *output* (*e.g.* from a pretty printer) and a span for the node. The *splice* function is applied to this triple, to compute a fragment of output source text by splicing together existing source text with any newly produced source text.

The refactoring type controls which part of the input source is used for the output and how much is discarded by *splice*. The *splice* function is given in Figure 2, which has the following three cases depending on refactoring type:

- Replace - the output source for the current context is the source text up to the lower bound of the node (*pre*) concatenated with the reprinting *output*. The input source between the lower and upper bounds is discarded.

- After - the output source is the input source up to the node's upper bound (*pre*), concatenated with the reprinting *output*.
- Before - the output is the source text up to the lower bound of the node (*pre*), concatenated with the reprinting *output* and then concatenated with the input source text between the lower and upper bounds of the node (*post*).

In each case, the cursor is updated to be the upper bound of the refactored node. Splicing uses the *splitBySpan* function of type:

```
splitBySpan :: Span → Source → (Source, Source)
```

Given a lower bound and upper bound pair of positions, *splitBySpan* splits a Source into a prefix and suffix, where the prefix is of the length of source from the upper bound minus the lower bound. That is, the lower bound position is taken as the start of the parameter source and the source is split into two at the upper bound.

The third and final step of *enter* is to navigate to the right sibling by *go right*, producing the source text *outputSib*. The result of *enter* is then the concatenation of the output from the current node or its children (*output*) with the output from the right sibling (*outputSib*).

Thus, *enter* computes a depth-first traversal of the AST, and simultaneously a linear traversal of the source text (see Corollary 1).

Top-level. The top-level function of the reprint algorithm converts an incoming data type to the datatype generic zipper, and enters into the root node, setting the cursor at the start of the file:

```
reprint :: (Monad m, Data ast)
   ⇒ Reprinting m → ast → Source → m Source
reprint reprinting ast input
  -- If the input is empty return empty
   | Text.null input = return mempty
  -- Otherwise proceed with the algorithm
   | otherwise = do
    -- Initial state comprises start cursor and input source
    let state0 = (initPosition, input)
    -- Enter the top-node of a zipper for 'ast'
    let comp = enter reprinting (toZipper ast)
    (out, (_, remaining)) ← runStateT comp state0
    -- Add to the output source the remaining input source
    return (out <> remaining)
```

Note that the final output is the concatenation of the output source from *enter* with the remaining input source text.

3.3 Reprinting parameter functions

The well-formedness conditions on an AST (Definition 1) require refactored nodes to have a source span. This is captured by the following class, Refactorable:

```
class Refactorable t where
  isRefactored :: t → Maybe RefactorType
  getSpan      :: t → Span
```

That is, refactorable data types provide a span, and also information on whether they have been refactored using the RefactorType (Section 3.1), where Nothing means that a node has not been refactored.

The *reprint* algorithm does not directly enforce the Refactorable constraint since this does not interact well with the datatype generic implementations in GHC Haskell (see discussion in Section 6.2). Instead, we provide the following builder function for generating a reprinting for a Refactorable type:

```
splice :: Monad m ⇒ (RefactorType, Source, Span)
   → StateT (Position, Source) m Source
splice (typ, output, (lb, ub)) = do
  (cursor, inp) ← get
  case typ of
    Replace → do
      -- Get source up to start of refactored node
      let (pre, inp′) = splitBySpan (cursor, lb) inp
      -- Remove source covered by refactoring
      let (_, inp′′) = splitBySpan (lb, ub) inp′
      put (ub, inp′′)
      return (pre <> output)
    After → do
      -- Get source up to end of the refactored node
      let (pre, inp′) = splitBySpan (cursor, ub) inp
      put (ub, inp′)
      return (pre <> output)
    Before → do
      -- Get source up to start of refactored node
      let (pre, inp′) = splitBySpan (cursor, lb) inp
      -- Discard portion consumed by the refactoring
      let (post, inp′′) = splitBySpan (lb, ub) inp′
      put (ub, inp′′)
      return (pre <> output <> post)
```

Figure 2: Splicing together refactored text and input text.

```
genReprinting :: (Monad m, Refactorable t, Typeable t)
   ⇒ (t → m Source)
   → t → m (Maybe (RefactorType, Source, Span))
genReprinting f z = do
  case isRefactored z of
    Nothing          → return Nothing
    Just refactorType → do
      output ← f z
      return (Just (refactorType, output, getSpan z))
```

Given a function *f* that converts some refactorable type *t* to some source text, *genReprinting* wraps *f* and the methods of the Refactorable class to producing a Reprinting-typed function.

A function *catchAll* provides a default generic query which can be used to construct a generic reprinting:

```
catchAll :: Monad m ⇒ a → m (Maybe b)
catchAll _ = return Nothing
```

For example, given a monomorphic function *repr* :: S → Source on some syntax type S which is Refactorable, then a generic reprinting can be defined by:

```
reprinting :: Reprinting Identity
reprinting = catchAll ‘extQ‘ (genReprinting (return ∘ repr))
```

where *extQ* :: (Typeable *a*, Typeable *b*) ⇒ (*a* → *q*) → (*b* → *q*) → *a* → *q* provides extension of a generic query from the *Scrap Your Boilerplate* library [8]. Here we use the Identity monad, *i.e.*, the reprinting is pure.

3.4 Example

The introduction gave our running example in a simple SSA-like language with assignments and integer addition which we use to illustrate the reprinting algorithm. The complete source for the examples is part of the library on Hackage.[1]

The example language is defined using the following data types to capture the AST, providing source code spans in each node and a boolean flag only in those parts of the AST that are subject to our refactoring (the Expr type):

```
type AST = [Decl]
data Decl = Decl Span String Expr
  deriving (Data, Typeable)
data Expr =
     Plus Bool Span Expr Expr
   | Var Bool Span String
   | Const Bool Span Int
  deriving (Data, Typeable)
```

Note that the `DeriveDataTypeable` GHC language extension is required to derive the Data and Typeable classes which are needed for the generic zipper.

We define a simple parser for the language (not included here), providing the function *parse*::Source → AST which sets the boolean flag of expressions to False, indicating no refactoring has been performed yet.

We thus have an instance of Refactorable for expressions:

```
instance Refactorable Expr where
  isRefactored (Plus True _ _ _) = Just Replace
  isRefactored (Var True _ _)    = Just Replace
  isRefactored (Const True _ _)  = Just Replace
  isRefactored _                 = Nothing
  getSpan (Plus _ s _ _) = s
  getSpan (Var _ s _)    = s
  getSpan (Const _ s _)  = s
```

Note that explicit cases have to be written only for those AST nodes which are subject to refactoring.

Given a simple pretty printer for expressions (not included here) of type *prettyExpr* :: Expr → Source, we define a reprinter for refactored expressions (but not the full AST data type of declarations) using the *genReprinting* helper:

```
exprReprinter :: Reprinting Identity
exprReprinter = catchAll ‘extQ‘ reprintExpr
  where reprintExpr x =
          genReprinting (return ∘ prettyExpr) (x :: Expr)
```

The function *refactorZero* :: AST → AST performs our desired refactoring by removing Plus with one side zero and annotating the other subexpression as refactored.

Finally, we put all the components together to parse, refactor, and reprint:

```
refactor :: Source → Source
refactor input = runIdentity
  ∘ (λast → reprint exprReprinter ast input)
  ∘ refactorZero
  ∘ parse
  $ input
```

We can now run the example as follows:

```
input = "x = +(1,2)\n"
"y = +(x,0)\n"
"// Calculate z\n"
"z = +( 1, +(+(0,x) ,y) )\n"
output = (putStrLn ∘ Text.unpack ∘ refactor) input
```

Running *output* prints the refactored source to stdout:

```
*Main> output
x  =  +(1,2)
y  =  x
// Calculate z
z  =  +( 1,  +(x ,y) )
```

Example using "After". As a final example, we show the use of the After reprinting style as well as a monadic reprinter. In our example language it is possible for every variable declaration to be calculated beforehand, *i.e.*, all programs are terminating. Using our reprinter we will pass over the AST, calculate variable values and comment declarations with the resulting value. For our example, this produces:

```
x = +(1,2) // x = 3
y  =  +(x,0) // y = 3
// Calculate z
z  =  +( 1,  +(+(0,x)  ,y) ) // z = 7
```

We first define an *eval* function that takes an expression, an environment (map from variables to values) and returns a Maybe Int wrapped in the State monad:

```
eval :: Expr → State [(String, Int)] (Maybe Int)
eval (Plus _ _ e1 e2) = do
  e1 ← eval e1
  e2 ← eval e2
  return (fmap (+) e1 < * > e2)
eval (Const _ _ i) = (return ∘ Just) i
eval (Var _ _ s) = do
  l ← get
  return (lookup s l)
```

If an unassigned variable is used then Nothing is returned, otherwise Just of the calculated value of the variable is returned (using the applicative machinery on Maybe).

We then define a reprinting that applies *eval* on Decl pieces of syntax, and returns an After refactoring if a value is calculated, producing a comment after each Decl node:

```
commentPrinter :: Reprinting (State [(String, Int)])
commentPrinter = catchAll ‘extQ‘ decl
```

[1] http://hackage.haskell.org/package/reprinter

```
  where
    decl (Decl s v e) = do
      val ← eval (e :: Expr)
      case val of
        Nothing → return Nothing
        Just val → do
          modify ((v, val):)
          let msg = " // " ++ v ++ " = " ++ show val
          return (Just (After, Text.pack msg, s))
```

Note that the State monad is also updated (via *modify*) to record the variable-value assignment of a declaration.

Using *reprint* and *parse*, we now define a Source to Source transformation which parses, refactors, and reprints:

```
refactor2 :: Source → Source
refactor2 input = flip evalState [ ]
  ∘ flip (reprint commentPrinter) input
  ∘ parse
  $ input

output2 = (putStrLn ∘ Text.unpack ∘ refactor2) exampleSource
```

where *output2* prints the refactoring result to stdout:

```
*Main> output2
x = +(1,2) // x = 3
y  =  +(x,0) // y = 3
// Calculate z
z  =  +( 1,  +(+(0,x)  ,y) ) // z = 7
```

4 REPRINTING AND PARSING AS A BIDIRECTIONAL LENS

A *bidirectional transformation* is a pair of programs converting data from one representation to another, and vice versa, often called the *source* and the *view*. A bidirectional *lens* is a bidirectional transformation capturing the notion of being able to update a source to produce a new source based on changes to a view [12].

Definition 2. A *bidirectional lens* comprises a *source type* S a *view type* V and a pair of **view** and **update** combinators typed [4]:

$$\mathbf{view} : S \to V$$

$$\mathbf{update} : V \times S \to S$$

A *well-behaved lens* satisfies the axioms:

$$\begin{aligned} \mathbf{view}(\mathbf{update}(v, s)) &\equiv v & (\textit{update-view}) \\ \mathbf{update}(\mathbf{view}(s), s) &\equiv s & (\textit{view-update}) \end{aligned}$$

The first says: *updates to a source with a view should be exactly captured in the source, such that viewing recovers the original view.* The second says: *updating with an unchanged view of a source does not change the source.*

Reprinting and parsing together form a bidirectional lens, which satisfies the (*update-view*) axiom.

Proposition 1 (Reprinting-parsing lens)**.** Let the source type S be the type of source text, and let the view type $V = AST$ – the top-level type of abstract syntax trees – with lens operations **view** = **parse** : $S \to AST$ and **update** = **reprint** : $AST \times S \to S$. This assumes that we have already specialised the reprinting algorithm with some parameter reprinting function. The (*update-view*) axiom of well-behaved lenses then holds:

$$\mathbf{reprint}(\mathbf{parse}(source), source) \equiv source$$

i.e., parsing to an AST and then reprinting this (unmodified) AST with the original source, yields the source. This holds if the parser satisfies the well-formedness condition (Definition 1).

Reprinting-parsing forms a *well-behaved* lens if the (*view-update*) axiom holds:

$$\mathbf{parse}(\mathbf{reprint}(ast, source)) \equiv ast$$

This implies that any pretty printing used by the reprinting parameterising the reprinter is the left-inverse of parsing (*i.e.*, pretty printing then parsing is the identity function). This should hold in any reasonable situation, so reprinting should form a well-behaved lens with parsing.

This lens perspective on parsing-reprinting provides programmers with a guide as to what properties to test and/or verify for their parsers and printing algorithms.

In the text-patching approach to reprinting by de Jonge *et al.*, a similar condition to (*update-view*) is introduced called *preservation* and (*view-update*) called *correctness* [3]. They also comment on the connection to lenses.

Well-behaved lenses are called *very well-behaved* if an additional property holds: an update followed by a second update is equivalent to just the second update. For reprinting, this would equate to the following axiom, which is unlikely to hold for most reprinters:

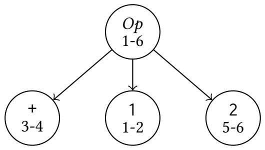
$$\mathbf{reprint}(ast_2, \mathbf{reprint}(ast_1, source)) \equiv \mathbf{reprint}(ast_2, source)$$

This would imply that any changes made to *source* by ast_1 are overwritten/subsumed by the changes in ast_2. Subsequently, source spans in ast_2 would also need to closely correspond to actual code in *source* and $\mathbf{reprint}(ast_1, source)$ simultaneously–which is unlikely.

5 RELAXING SOURCE COHERENCE

Recall from the definition of *source coherence* (Definition 1) that *sequentiality* requires siblings to have non-overlapping spans in increasing order. We define here the notion of *weak source coherence* as a generalisation of *source coherence* without the sequentiality condition *i.e.*, siblings need not be ordered according to their spans.

This arises when the shape of the AST data structure does not match the lexical shape of code. Consider, for example, a piece of infix syntax `1 + 2` which is parsed into a ternary tree node:



This AST structure violates the well-formedness condition of *sequentiality*, since the left-most child has a span that is greater than its right sibling. As a consequence, applying the previous reprinting algorithm to such an AST would provide a traversal of the nodes that is not in the order of source spans (*i.e.*, Corollary 1 no longer holds). If we refactored one of the first two children, then the reprinting algorithm would subsequently splice together text *out of order*, producing an incorrect output.

The above example however still provides source span information and still satisfies enclosure, and thus is *weak source coherent*. This is enough to provide an adapted reprinting algorithm.

We summarise here a variation of the reprinting algorithm of Section 3 which only requires ASTs to be weakly source coherent. Whilst this algorithm is more general, it has yet to be determined whether the adaptation stands up equally well in real-world use regarding runtime and memory performance. The *reprinter* library provides both algorithms.

Alternate algorithm. The essence of the generalised algorithm is to delay splicing, traversing the AST similarly to before (depth first) but gathering a list of refactoring and splicing information. This list is then sorted by the spans, providing a sequentialised list of refactorings/splicings, that is, in source span order. The *splice* function (as defined before in Figure 2) is applied to each of these and the resulting source fragments are concatenated.

The top-level *reprinter* function is as before, but we redefine *enter* as *enter′* in Figure 3 (both have the same type). This function proceeds in three steps.

Firstly, an AST zipper traversal is performed by the intermediate function *getRefactorings*. This function resembles closely the definition of *enter* in the original algorithm (Section 3) which performed the zipper traversal and source splicing in tandem. Instead, *getRefactorings* collects a list of results, delaying splicing, of type [(RefactorType, Source, Span)]. That is, triples of refactoring information, newly generated source text fragments, and the span of a node from which the fresh source text originates.

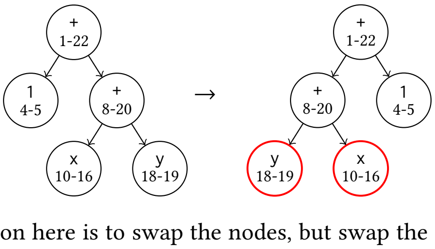
In the second step, we sort the resulting list of refactorings by their span (via the function *sortBySpan* below using *sortOn* from the module Data.List), applying *splice* to each element of the resulting list. The resulting fragments are then concatenated in the final step.

This algorithm subsumes the former since, for a source coherent AST, the sorting will be superfluous since the traversal of *getRefactorings* will produce a list already ordered by spans due to Corollary 1.

6 DISCUSSION

6.1 Considerations about well-formedness

The well-formedness condition (Definition 1) implies that transformations to an AST must take care with source span information. For example, transformations which commute nodes in the tree will almost certainly violate well-formedness, *e.g.*



The solution here is to swap the nodes, but swap the span information back, such that the span information stays in its original location in the tree, ensuring well formedness. There are other more complicated situations which might require more involved span recalculation as part of the transformation.

```
enter′ :: Monad m ⇒ Reprinting m → Zipper ast
        → StateT (Position, Source) m Source
enter′ reprinting zipper = do
    -- Step 1: Get refactorings via AST zipper traversal
    rs ← lift (getRefactorings reprinting zipper [ ])
    -- Step 2: Do the splicing on the sorted refactorings
    srcs ← mapM splice (sortBySpan rs)
    -- Step 3: Concatenate fragments
    return (Text.concat srcs)
  where
    sortBySpan = sortOn (λ(_, _, sp) → sp)

getRefactorings :: Monad m ⇒
         Reprinting m → Zipper ast
    →   [(RefactorType, Source, Span)]
    →   m [(RefactorType, Source, Span)]
getRefactorings reprinting zipper acc = do
    -- Step 1: Apply a refactoring
    refactoringInfo ← query reprinting zipper
    -- Step 2: Deal with refactored code or go to children
    acc ← case refactoringInfo of
      -- No refactoring; go to children
      Nothing → go down' acc
      -- A refactoring was applied, add it to the accumulator
      Just r → return (r : acc)
    -- Step 3: Enter the left sibling of the current focus
    acc ← go right acc
    -- Finally return the accumulated refactorings
    return acc
  where
    go direction acc =
      case direction zipper of
        -- Go to next node if there is one
        Just zipper → getRefactorings reprinting zipper acc
        -- Otherwise return the empty string
        Nothing → return acc
```

Figure 3: Generalised algorithm for weak source coherency.

6.2 Constrained generic programming

Section 3.3 defined the *genReprinting* function to wrap an output function of type $t \rightarrow m$ Source where Refactorable t into a function $t \rightarrow m$ (Maybe (RefactorType, Source, Span)), wrapping the methods of Refactorable. However, there is nothing to force the programmer to use *genReprinting* to define a Reprinting. Indeed, the last example of Section 3.4 defined a Reprinting by hand.

An alternate approach would be to define *reprint* directly in terms of Refactorable types, *e.g.*,

```
type Reprinting m =
  ∀b.(Typeable b, Refactorable b) ⇒
    b → m (Maybe (RefactorType, Source, (Position, Position)))
```

$$\begin{aligned}&\mathit{reprint} :: (\mathsf{Monad}\ m, \mathsf{Data}\ p, \mathsf{Refactorable}\ p)\\&\quad\Rightarrow \mathsf{Reprinting}\ m \to p \to \mathsf{Source} \to m\ \mathsf{Source}\end{aligned}$$

The core of the algorithm (*enter*, Figure 1, p. 5) could then be defined in terms of the methods of Refactorable directly. However, the Refactorable constraint must then be pushed into the generic zipper and the datatype generic operations, which are unconstrained. Much of the datatype generic infrastructure for GHC Haskell does not support this *constrained genericity*.

A potential solution is to parameterise datatype generic operations on additional constraint parameters (via GHC's constraint kinds). For example:

$$\begin{aligned}&\textbf{type}\ \mathsf{GenericCQ}\ (c :: * \to \mathsf{Constraint})\ r =\\&\qquad \forall a.(\mathsf{Data}\ a, c\ a) \Rightarrow a \to r\end{aligned}$$

and to define a constrained zipper type, *e.g.*, $\textbf{data}\ \mathsf{Zipper}\ (c :: * \to \mathsf{Constraint})\ a = \ldots$ which adds the constraint within the intermediate data types of the zipper.

We have done some early exploration and it seems plausible, though such constraints will need to propagated throughout the rest of the libraries. This is further work and would be useful far beyond the topic of this paper.

6.3 Concluding remarks

We have presented a general algorithm that provides core functionality for refactoring tools: outputting source text that preserves secondary notation in untransformed code. The algorithm is relatively short thanks to the GHC Haskell's datatype generic programming facilities. Such an implementation would have been much more complicated 15 years ago.

We have been using a variant of this algorithm for several years and it has proven robust in the context of a real tool (CamFort). In terms of asymptotic performance, the core traversal is $O(n)$. The absolute performance is degraded somewhat by the use of datatype generics which are notoriously slow. Recent work suggests how to improve this considerably via staging [15]. Exploring this, with performance benchmarks, is further work.

Acknowledgements. This work was supported by the EPSRC grant EP/M026124/1. Thank you to the rest of CamFort team: Mistral Contrastin, Matthew Danish, and Andrew Rice for input on the concepts in this paper. We also thank the School of Computing and Faculty of Sciences at the University of Kent for their financial support of the first two authors during the summer of 2017.

REFERENCES

[1] Michael D Adams. Scrap Your Zippers: a Generic Zipper for Heterogeneous Types. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 13–24. ACM, 2010.

[2] Mistral Contrastin, Matthew Danish, Dominic Orchard, and Andrew Rice. Lightning talk: Supporting Software Sustainability with Lightweight Specifications. In *Proceedings of the Fourth Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE4), University of Manchester, Manchester, UK, September 12–14*, volume 1686. CEUR Workshop Proceedings, 2016.

[3] Maartje de Jonge and Eelco Visser. An algorithm for layout preservation in refactoring transformations. *SLE*, 11:40–59, 2011.

[4] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices*, 40(1):233–246, 2005.

[5] Gérard Huet. The zipper. *Journal of functional programming*, 7(5):549–554, 1997.

[6] Róbert Kitlei, László Lóvei, Tamás Nagy, Zoltán Horváth, and Tamás Kozsik. Layout preserving parser for refactoring in Erlang. *Acta Electrotechnica et Informatica*, 9(3):54–63, 2009.

[7] Jan Kort and Ralf Lammel. Parse-tree annotations meet re-engineering concerns. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 161–170. IEEE, 2003.

[8] Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM.

[9] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ACM SIGPLAN Notices*, volume 40, pages 204–215. ACM, 2005.

[10] Conor McBride. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript*, pages 74–88, 2001.

[11] Dominic Orchard and Andrew Rice. Upgrading Fortran Source Code using Automatic Refactoring. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, pages 29–32, 2013.

[12] B.C Pierce. The weird world of bi-directional programming, 2006. ETAPS invited talk, slides available from http://www.cis.upenn.edu/~bcpierce/papers/lenses-etapsslides.pdf.

[13] Michael L Van De Vanter. Preserving the documentary structure of source code in language-based transformation tools. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 131–141. IEEE, 2001.

[14] Mark GJ van den Brand and Jurgen J Vinju. Rewriting with Layout. In *Proceedings of RULE*, 2000.

[15] Jeremy Yallop. Staging generic programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 85–96. ACM, 2016.