Scrap Your Reprinter
A Datatype Generic Algorithm for Layout-Preserving Refactoring

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

Dominic Orchard
School of Computing University of Kent
da.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, and 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 are free to discard this information, since it is irrelevant for generating the final binary output. 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:

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 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 \(+(e, 0) \leadsto e\) and \((0, e) \leadsto e\) 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 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. Section 5 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:

\[ + ( 1, + ( + (0, x), y) ) \]

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

\[
\begin{array}{c|cccccccccccccccccccc}
\text{pre} & + ( 1, + (0, x), y) & \rightarrow & + ( 1, + (x, y) ) \\
\text{post} & + ( 1, + (0, x), y) & \rightarrow & + ( 1, + (x, y) ) \\
\end{array}
\]

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.

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

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

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 code. The cursor is also updated to the lower bound of the node:

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

```
+---+---+---+
|   |   |   |
+---+---+---+
  1 5 8
+---+---+---+
  y 10-16 x 18-19
```

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

```
+---+---+---+
|   |   |   |
+---+---+---+
  1 5 8
+---+---+---+
  y 10-16 x 18-19
```

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

```
+---+---+---+
|   |   |   |
+---+---+---+
  1 5 8
+---+---+---+
  y 10-16 x 18-19
```

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 – the context of the focus. The focus is shifted by navigation operations. Our algorithm is defined in terms of a generic zipper construction (Scrap Your Zipper [1]) to provide datatype generic traversal.

2.1 Standard Zippers

Zippers make traversing and transforming a data structure easier, as well as providing more efficient operations, such as constant-time updates. Suppose you had the following list:

```
let list = [1, 2, 3]
```

Without a zipper, a simple recursive function, traversing the list by deconstructing it one constructor at a time, will lose information about previous elements. Zippers retain this information as part of the data type should we need it, either for reconstruction of the original data, or for contextual operations.

A zipper is made up of two parts:

- **Focus** - This is the part of the data structure that we are currently viewing.
- **Context** - This represents the rest of our data type not already contained in the focus. To represent this efficiently, nodes are represented by their siblings and their parent, so the tree’s direction is reversed within the context, with a path pointing all the way up to the root node.

A list zipper can be defined like so:

```
type ListZipper a = (ListFocus a, ListContext a)
type ListFocus a = [a]
data ListContext a
  = Root | Node a (ListContext a)
toListZipper :: [a] -> ListZipper a
toListZipper l = (l, Root)
```

When we first create a zipper, the focus represents the entire data structure (or in our case, the entire list), as we are at the root node, and the context is just `Root`.

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, Node x ys)
listDown ([], ys) = Nothing

listUp :: ListZipper a -> Maybe (ListZipper a)
listUp (xs, Node y ys) = Just (y : xs, ys)
listUp (xs, Root) = 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 `Root` is reached, we are at the top/beginning of the list, and there is nowhere left for us to go.

Thus, for the list `[1, 2, 3]`, we get the following zipper navigation:

```
(listDown 

(listUp (([1, 2, 3], Root)

(listUp ([(2, 3), Node 1 Root])

(listUp ([(3], Node 2 (Node 1 Root)))

(listUp ([(], Node 3 (Node 2 (Node 1 Root))))
```

Zippers on trees have a similar structure. 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 = (TreeFocus a, TreeContext a)
type TreeFocus a = Tree a
data TreeContext a
  = TRoot | TNode (TreeContext a) [Tree a] [Tree a]
```

A zipper representation of a datatype can be calculated from the algebraic representation of a datatype via Leibniz differentiation [10]. This implies that zippers can be automatically generated for any datatype. This enables generic zippers.

2.2 Generic zippers

Scrap Your Zipper (SYZ) [1] is a library that allows the programmer to traverse any tree structure with ease by 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 of SYZ in purple to make it clear where they are used in the algorithm.

A function `toZipper :: Data a ⇒ a → Zipper a` maps any Data type into the zipper representation. Our algorithm then only needs two of the zipper navigation operations provided by the SYZ library: `down` and `right`. The `down` operation moves the focus of the zipper to the leftmost child of the current node, of type:

```
[45x191]+
```

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 is at the root:

```
1 1-2 2-10

down
1 1-2 10-16
```

We also use `right`, which moves the focus of the zipper to the sibling to the right:

```
+ 1-2 2-10
  2 10-16
	right
  + 1-2 2-10
  2 10-16
```

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 function which is defined for all types `c` satisfying `Data` to produce a value of type `b` is applied to the current focus of the zipper to produce a `b` value.

### 3 Datatype generic reprinting; implementation

The reprinting algorithm stitches together an AST and a piece of source text into an output source text by a simultaneous traversal of both the AST and input source. The AST is traversed in depth-first order, and the input source is traversed linearly. This relies on the AST containing some information about the origin of its nodes in the original source file, along with some well-formedness conditions of this location information. Note that the datatype’s definition for an AST usually spans many mutually recursive datatypes.

**Definition 1.** An AST is source coherent if every refactoring 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 is less than or equal to the lower-bound position of its right sibling (if there is one).

These two conditions hold for the example here, which is defined in full below; the AST is source coherent. 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, though there may be cycles with the same upper and lower bounds not ruled out by its condition.

We split our description of the algorithm into three parts: the main types (Section 3.1), the core of the algorithm with the top-level function `reprint` and the intermediates `enter`, `enterDown` and `enterRight` (Section 3.2), and finally helpers for building “reprintings” (generic functions parameterising `reprint`) (Section 3.3).

The final section provides an implementation of the introduction code (Section 3.4).

#### 3.1 Types

Let `Source` be the type of source text, for which we use the efficient `ByteString` representation:

```
import qualified Data.ByteString.Char8 as B
```

```
type Source = B.ByteString
```

Positions in source code are given by pairs of column and line numbers (integers), which are 1-indexed:

```
data Position = Position { posColumn :: Int, posLine :: Int }
```

```
deriving Data
```

```
initPosition = Position { posColumn = 1, posLine = 1 }
```

The span of a piece of source text can be represented by a pair of positions.

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

The top-level function `reprint` has type:

```
reprint :: (Monad m, Data p) ⇒ Reprinting m → p → 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 = ∀h, Typeable b ⇒ b → m (Maybe (RefactorType, Source, Span))
data RefactorType = Before | After | Replace
```

Thus, a reprinting maps any `Typeable` type `b` to `Maybe` of a triple of the refactoring type, the new source text for this node, and a pair of positions (`Span`), which represent the lower and upper bound positions of the node. We provide some helper functions for constructing a `Reprinting` in Section 3.3.
Note that \textit{reprint} and \textit{Reprinting} are parameterised by some monad \textit{m}. This provides extra power if an application wishes to include some additional 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 another example in Section 3.4.

3.2 Zipper traversal; core

The zipper traversal part of the algorithm is performed by a trio of functions: \textit{enter}, \textit{enterDown}, and \textit{enterRight}, of type:

\begin{align*}
\text{enter, enterDown, enterRight} & : \text{Monad } m \Rightarrow \\
\text{Reprinting } m & \rightarrow \text{Zipper } p \rightarrow \text{StateT} \ (\text{Position, Source}) \ m \ Source
\end{align*}

where \text{Zipper} provides the zipper for some type \textit{p}, provided by the Scrap Your Zipper library [1]. The implementation is shown in Figure 1. Recall that the zipper operations are in \textit{purple}.

The \text{enterDown} and \text{enterRight} are respectively used to navigate to a node’s children and to its siblings, using the navigation operations of the generic zipper.

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

The second step matches on this information returned from the reprinting to determine whether to perform some splicing of the input source text or to navigate to the children. If \textit{refactoringInfo} is \textit{Nothing} then \text{enterDown} is called to proceed to the children. Otherwise, we have \text{Just (\textit{typ, output, (lb, ub)})} indicating a refactored node with \textit{refactoring} \textit{typ}, new output source \textit{output} (e.g., from a pretty printer), and a pair of lower bound \textit{(lb)} and upper bound \textit{(ub)} positions for the node.

The \textit{refactoring} type controls how much of the input source is used for the output and how much of the input source is discarded:

- \text{Replace} - the output source for the current context is the source text up to the lower bound of the node concatenated with the reprinting output. The input source between the lower and upper bounds is discarded.
- \text{After} - the output source is the input source up to the node’s upper bound, concatenated with the reprinting output.
- \text{Before} - the output text up to the lower bound of the node, concatenated with the reprinting output and then concatenated with the input source text between the lower and upper bounds of the node.

In each case, the cursor is updated to be the upper bound of the refactored node.

The third and final step is to navigate to the right sibling, producing the source text \textit{output}’. The result of \textit{enter} is then the concatenation of the output from the either current node or its children \textit{(output)} with the output from the right sibling \textit{(output’)}.

The algorithm makes use of the \text{splitBySpan} function of type:

\begin{align*}
\text{splitBySpan} & : \text{Span} \rightarrow \text{Source} \rightarrow \ (\text{Source, Source})
\end{align*}

Given a lower bound and upper bound pair of positions, \text{splitBySpan} splits a \text{Source} into a prefix and suffix, where the prefix is of the length of source from the lower bound minus the lower bound. That

\begin{align*}
\text{enter, enterDown, enterRight} & : \text{Monad } m \Rightarrow \\
\text{Reprinting } m & \rightarrow \text{Zipper } p \rightarrow \text{StateT} \ (\text{Position, Source}) \ m \ Source
\end{align*}

\begin{align*}
\text{enterDown } f \ z = \\
\text{case } \text{down’ } z \text{ of} \\
& \text{Just } dz \rightarrow \text{enter } f \ dz \quad \text{-- Go to children} \\
& \text{Nothing} \rightarrow \text{return } \text{B.empty} \quad \text{-- No children}
\end{align*}

\begin{align*}
\text{enterRight } f \ z = \\
\text{case } \text{right } z \text{ of} \\
& \text{Just } rz \rightarrow \text{enter } f \ rz \quad \text{-- Go to right sibling} \\
& \text{Nothing} \rightarrow \text{return } \text{B.empty} \quad \text{-- No right sibling}
\end{align*}

\begin{figure}[h]
\begin{center}
\begin{tabular}{l}
\text{Figure 1. Core traversal of the reprinting algorithm}
\end{tabular}
\end{center}
\end{figure}

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.
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 p)
  ⇒ Reprinting m → p → Source → m Source
reprint reprinting tree input
  -- If the input is null then null is returned
  | B.null input = return B.empty
  -- Otherwise go with the normal algorithm
  | otherwise = do
    -- Initial state comprises start cursor and input source
    let state0 = (initPosition, input)
    -- Enter the top-node of a zipper for 'tree'
    let comp = enter reprinting (toZipper tree)
    (out, (remaining)) ← runStateT comp state0
    -- Add to the output source the remaining input source
    return $ out 'app' 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) requires 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 implies 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 5.2). Instead, we provide the following builder function for generating a reprinting for a Refactorable type:

```haskell
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
      getSpan $ Just (refactorType, output, getSpan z)
```

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

A function \( \text{catchAll} \) 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 \( \text{repr} :: S → Source \) 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 \( \text{extQ} :: (Typeable a, Typeable b) ⇒ (a → q) → (b → q) → a → r \) 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 showed a simple SSA-like language with assignments and integer addition (with prefix operations). We use the following data types to capture its AST:

```
data AST a =
  Seq a (Decl a) (AST a)
  | Nil a
                           deriving (Data, Typeable)
data Decl a =
  Decl a Span String (Expr a)
                           deriving (Data, Typeable)
data Expr a =
  Plus a Span String (Expr a)
  | Var a Span String
  | Const a Span Int
                           deriving (Data, Typeable)
```

Note that the data types derive the Data and Typeable classes to provide the datatype generic facilities.\(^1\) These datatypes provide source code spans in each node and an annotation type \( a \) which can be used for indicating which nodes have been refactored.

We define a simple parser (not included here), which provides the function \( \text{parse} :: \text{Source} → AST \text{ Bool} \) which annotates each with part of the syntax tree with False indicating no refactoring has been performed yet.

We thus have an instance of Refactorable for expressions:

```
instance Refactorable (Expr Bool) 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
```

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

```
exprReprinter :: Reprinting Identity
exprReprinter = catchAll 'extQ' printExpr
  where reprExpr x =
    genReprinting (return . prettyExpr) (x :: Expr Bool)
```

\(^1\)Note that this requires the \text{returnDataTypeable} language extension in GHC.
We first define an output variable declaration to be calculated before-hand, producing a comment after each Decl syntax, and returns an Maybe Int variable.

\[
\begin{align*}
\text{eval} &:: \text{Expr Bool} \\
\text{val} &\leftarrow \text{eval} (e :: \text{Expr Bool})
\end{align*}
\]

where

\[
\begin{align*}
\text{decl} (\text{Decl} \_ s v e) &= \text{do} \\
\text{val} &\leftarrow \text{eval} (e :: \text{Expr Bool})
\end{align*}
\]

\[
\begin{align*}
\text{case} \text{val} \text{of} \\
\text{Nothing} &\rightarrow \text{return Nothing} \\
\text{Just} \text{val} &\rightarrow \text{do} \\
\text{modify} &\left(\langle v, \text{val}\rangle\rangle\right)
\end{align*}
\]

\[
\begin{align*}
\text{let} \ \text{msg} &\equiv " \ " + v + " = " + \text{show val} \\
\text{return} &\left(\langle \text{After, B.pack}\rangle \text{msg}, \text{x}\rangle\right)
\end{align*}
\]

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

Using \text{reprint} and \text{parse}, we now define a Source to Source transformation:

\[
\begin{align*}
\text{reprint2} &:: \text{Source} \rightarrow \text{Source} \\
\text{reprint2} \text{input} &= \langle \text{flip (reprint commentPrinter)} \rangle \text{input} \\
&\circ \text{parse} \circ \text{input}
\end{align*}
\]

where \text{output2} prints out the result on stdout:

\[
\begin{align*}
\text{Main}> \text{output2} \\
x &= +(1,2) \\
y &= +x,0 \\
// Calculate z \\
z &= +( 1, +(+(0,x) ,y) ) \ \\
\end{align*}
\]

\[
\begin{align*}
\text{z} &= 7 \ \\
\end{align*}
\]

\[
\begin{align*}
\text{4 Reprinting and parsing as a bidirectional lens}
\end{align*}
\]

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 \text{view} and \text{update} combinators typed [4]:

\[
\begin{align*}
\text{view} : S \rightarrow V \\
\text{update} : V \times S \rightarrow S
\end{align*}
\]

A well-behaved lens satisfies the axioms:

\[
\begin{align*}
\text{view(update(v, s))} &\equiv v &\quad \text{(update-view)} \\
\text{view(update(v, s))} &\equiv s &\quad \text{(view-update)}
\end{align*}
\]

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 \text{(update-view)} axiom.

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

\[
\begin{align*}
\text{reprint(view(source), source)} &\equiv \text{source}
\end{align*}
\]
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).

But can reprinting–parsing be a well-behaved lens? The \((\text{view-update})\) axiom would be:

\[
\text{parse} (\text{reprint} (\ast_t, \text{source})) \equiv \ast_t
\]

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

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

Note that 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:

\[
\text{reprint} (\ast_{t_2}, \text{reprint} (\ast_{t_1}, \text{source})) \equiv \text{reprint} (\ast_{t_2}, \text{source})
\]

This would imply that any changes made to source by \(\ast_{t_1}\) are erase/subsumed by the changes in \(\ast_{t_2}\). Subsequently, source spans in \(\ast_{t_2}\) would also need to closely correspond to actual code in source and \(\text{reprint} (\ast_{t_1}, \text{source})\) simultaneously— which is unlikely.

5 Discussion

5.1 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, \(e.g.,\)

\[
\begin{align*}
1 & \rightarrow 1 \times 22 \\
5 & \rightarrow 5 \times 20 \\
9 & \rightarrow 9 \times 18-19 \\
10-16 & \rightarrow 10-16
\end{align*}
\]

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.

Another issue occurs if the shape of the AST data structure does not match the lexical shape of code. For example, consider a piece of infix syntax \(1 + 2\) which is parsed into a ternary tree node:

\[
\begin{align*}
\ast & \rightarrow \ast \\
1 & \rightarrow 1 \times 2 \\
2 & \rightarrow 2 \times 3-4 \\
5-6 & \rightarrow 5-6
\end{align*}
\]

This parsing violates the well-formedness condition of sequentiality.

To solve this, we could include traversal information in the \(\text{Refactorable}\) class with the parent specifying which order its children should be looked at. Or we could look at the existing span lower bounds of children, and choose the order to view them based on the order they appear in the source text. Adapting our algorithm to deal with this is future work.

5.2 Constrained generic programming

Section 3.3 defined the \(\text{genReprinting}\) function to wrap an output function of type \(t \rightarrow m \text{Source}\) where \(\text{Refa}\) function \(t \rightarrow m\) \((\text{Maybe} (\text{Refactor}\text{Type}, \text{Source}, \text{Span}))\), wrapping the methods of \(\text{Refa}\). However, there is nothing to force the programmer to use \(\text{genReprinting}\) to define a \(\text{Reprinting}\). Indeed, the last example of Section 3.4 defined a \(\text{Reprinting}\) by hand.

An alternate approach would be to define \(\text{reprint}\) directly in terms of \(\text{Refactorable}\) types, \(\text{e.g.,}\)

\[
\begin{align*}
\text{type Reprinting } m = \\
\forall b. (\text{Typeable } b, \text{Refactorable } b) \Rightarrow \\
\quad b \rightarrow m \ \text{(Maybe} (\text{Refactor}\text{Type}, \text{Source}, (\text{Position}, \text{Position}))\text{)} \\
\forall m. \text{Reprinting } m \rightarrow p \rightarrow \text{Source} \rightarrow m \text{Source}
\end{align*}
\]

The core of the algorithm \((\text{enter}, \text{Figure 1, p. 5})\) could then be defined in terms of the methods of \(\text{Refactorable}\) directly. However, the \(\text{Refactorable}\) 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 \(\text{constrained}\) \(\text{genericity}\).

A potential solution is to parameterise datatype generic operations on additional constraint parameters (via GHC’s constraint kinds), \(\text{e.g.,}\)

\[
\begin{align*}
\text{type GenericCQ} \ (c :: * \rightarrow \text{Constraint}) \ r = \\
\forall a. (\text{Data } a, c a) \Rightarrow a \rightarrow r
\end{align*}
\]

and to define a constrained zipper type, \(\text{e.g.,}\) \(\text{data Zipper} \ (c :: * \rightarrow \text{Constraint}) \ a = \ldots\) 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.

5.3 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 is supported by the EPSRC grant EP/M026124/1. We also thank the School of Computing and Faculty of Sciences at the University of Kent for their financial support of the first author. Thanks to Vilem-Benjamin Liepelt who contributed to the implementation of the \(\text{splitBySpan}\) function.
References


