A Pretty Good Formatting Pipeline

Anya Helene Bagge and Tero Hasu

Bergen Language Design Laboratory
Dept. of Informatics, University of Bergen, Norway

Abstract. Proper formatting makes the structure of a program apparent and aids program comprehension. The need to format code arises in code generation and transformation, as well as in normal reading and editing situations. Commonly used pretty-printing tools in transformation frameworks provide an easy way to produce indented code that is fairly readable for humans, without reaching the level of purpose-built reformatting tools, such as those built into IDEs. This paper presents a library of pluggable components, built to support style-based formatting and reformatting of code, and to enable further experimentation with code formatting.

1 Introduction

Pretty-printing and code formatting are fundamental in the software language engineering toolbox. There are two main aspects to pretty-printing: creating textual output from an internal representation, and ensuring that the textual representation is visually pleasing and/or structurally clear. The ideal code formatting results in text that conveys the semantics as clearly as possible within the constraints of the medium.

In this paper, we will focus mainly on the formatting aspect, rather than on producing text output from an internal representation. Code formatting must take into account the syntactic and maybe also the semantic structure of the code in order to maximise the readability of the output. For example, the treatment of spacing around a minus symbol depends on whether it occurs as a binary operator (typically spaced on both sides), a unary operator (typically with no space after), or in some other context. For this reason, the input to the formatting must either contain the necessary syntactic and semantic information, or it must be reconstructed prior to formatting (via lexical, syntactic and/or semantic analysis).

We may consider several formatting concerns:

- *Horizontal spacing.* Good placement of spaces can make the details easier to grasp. For example, an expression such as `a+x * y` may easily confuse a casual reader as to the operator precedence, compared to `a + x*y` or even the neutrally spaced `a + x * y` (assuming normal operator priorities).
- *Indentation* has long been considered important to visual recognition of scope and nesting structures in program text [12]. This insight goes back at least to the 1960s [11].

- *Line breaking* may be necessary to fit the program to the screen or output medium. Line breaking must trade off efficient use of the human field of vision against program comprehension concerns such as keeping related things together and less related things further apart.
- Other processing, such as colourising or highlighting, attaching additional information such as hover text for HTML output and so on.

This paper describes our *pretty good formatter* (PGF) and associated experimental formatting components. PGF uses a pipeline of connected components to format or reformat code, where the various concerns of producing pretty code are separated into different *processors*; one for inserting horizontal space, one for breaking lines, one for adding colour and so on. PGF is designed to be useful for code generation, reformatting, or just reindenting code – depending on which components are plugged into the pipeline. PGF provides basic building blocks for making processing components, and the included components are designed to be reusable for different languages, with appropriate customisation.

The pipeline architecture itself is reusable for other purposes, and offers support for processing data concurrently in a pipeline, one data item at a time. Each processing component has access to a stream of input and a stream of output, and may specify a desired level of look-ahead and output history, in order to process based on a sliding window of information.

PGF is implemented as a reusable Java library, with experiments and prototypes in the Rascal meta-programming language [10] and the Scheme variant Racket [4]. The paper presents a mixture of the library proper, and the associated experiments. All source code and related content is online.[1]

The contributions of this paper include:

- *Pipeline processing*: a pipelined framework for building flexible code formatters and rule-based token processors (Section 2);
- *Line breaking*: a new heuristic line-breaking algorithm, and a reformulation of two other algorithms to fit our architecture (Section 3).
- *Pipeline plumbing*: a stream-based plugin architecture for connecting components in a pipeline, and a technique for grouping tokens and building pipelines dynamically (Section 4)

After presenting the contributions, the paper continues with a discussion and related work (Section 5), and conclusion (Section 6).

2 The Formatting Pipeline

Our formatter is built from a pipeline of *token processors*. Each processor receives a stream of tokens, processes them one by one, and feeds them to the next processor. The pipeline is illustrated in Fig. 1, which shows an input tree, an output text and two different pipeline arrangements, one doing line breaking and another one doing colourising and indentation. More details about the pipeline infrastructure is provided in Section 4.

[1] `http://nuthatchery.org/pgf/`

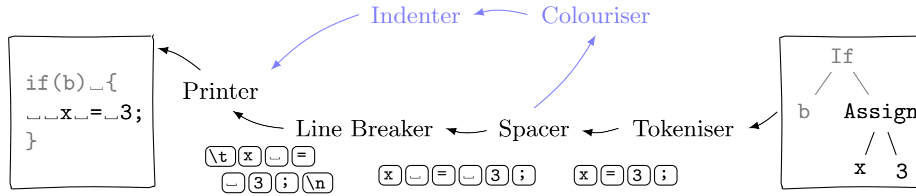

Fig. 1. A PGF pipeline. Starting with a program tree (rightmost), we translate it to a stream of tokens, then insert spaces, do indentation and line breaking, before producing program text (leftmost). The alternative path (light blue), adds colour and does indentation without line breaking. The token stream is shown below each step. In this example, each token is only one character wide; this is not generally the case.

2.1 Tokens and Categories

Tokens are either *data tokens* or *control tokens*. Each data token represents an atomic piece of source code to be output, while the control tokens (if used) may convey information about nesting, indentation levels, etc.

Each data token contains a string of characters, and is associated with a *category*, and possibly other metadata. The categories are user-definable, but we have selected a set of default categories (see Fig. 2), in order to aid reusability of components.

In this paper, we write data tokens enclosed in quotes, optionally followed by a colon and the category; for example, `"if":KEYWORD`, `" ":WS`, `"(":LPAR`. Control tokens are marked by a hash character followed by the category and possibly a list of parameters, e.g., `#BEGIN(expr)`. The PGF library itself uses slightly different conventions, depending on the implementation language.

Each token has only one specified category, but each category may be a subcategory of zero or more supercategories, with `A <: B` indicating that `A` is a subcategory of B – implying that any token categorised as `A` can also be categorised as `B`:

$$\frac{t : A \qquad A <: B}{t : B}$$

We will use `A <: B` also for the case where `A` is connected to `B` through one or more intermediate steps. Note that, in this case, there may be multiple paths through the supercategory chain, since each category can have multiple supercategories.

The use of categories is crucial for achieving reusability and language independence of processors. For example, languages that mostly follow the same spacing rules can reuse the same processor as long as the language-specific tokens are mapped to the same, general categories – possibly with a few additional language-specific rules.

When customising a processor for a particular language, it may be useful to have a one-to-one correspondence between token text and category, for instance, in the case of keywords. So, we may have `"if":IF` and `"else":ELSE`, with `{IF, ELSE} <:KEYWORD`, and a rule stating that we should never break the line between `IF` and `ELSE`.

```
category { TXT, SPC, CTRL } <: TOKEN;
category { START, STOP, BEGIN, END } <: CTRL;
category { WS, NL, COM } <: SPC; // horizontal, vertical space; comments
category { KEYWORD, PUNCT, ID, LITERAL, OP, GRP } <: TXT; // non−space
category { PAR, BRC, BRT } <: GROUPING; // parens, braces, brackets
category { LPAR, RPAR } <: PAR; // parentheses
category { LPAR, LBRC, LBRT } <: LGROUPING; // left grouping tokens
category { COMMA, SEMI, COLON, DOT } <: PUNCT; // punctuation
```

Fig. 2. A selection of the default token categories. In the notation, `category` defines one or more new categories (left-hand side), listing any supercategories to the right of the `<:` sign.

Token processors make their decisions primarily based on token categories, and not by inspecting token data. The flexible category hierarchy allows several aspects to be encoded. For example, with a default categorisation of `"("` as `LPAR`, a left parenthesis token will belong to both its own category, as well as `PAR` (parentheses in general), `LGRP` (left grouping tokens), `GRP` (groupings in general), `TXT` (printable, non-space tokens), `TOKEN` (any token), and decisions may be made on the basis of any of these.

2.2 The Tokeniser

The first processor in a pipeline is the *tokeniser*, which turns the input (whatever that might be) into a stream of tokens. We envisage three kinds of input:

– a parse tree, containing both structural and lexical information, or
– an abstract syntax tree, containing mostly structural information (maybe including semantic information), or
– a token sequence from a lexer, containing mostly lexical information.

With an abstract syntax tree, the lexical information must be reconstructed, for example using a pretty-print table generated from the grammar, or by a hand-written tokeniser which traverses the AST and outputs appropriate tokens for each AST node.

Lexer output might be more or less directly usable, possibly with mapping of the lexer's token categories to the formatter's. As we've only been using scannerless parsers, we have not explored this option further.

With a parse tree as input, all necessary information for printing and formatting should be available, and we can use a generic parse tree tokeniser to obtain the token stream. We have written such tokenisers that accepts parse trees in the UPTR and AsFix2 formats.[2]

[2] UPTR is the *Universal Parse Tree Representation*; it is used by Rascal, and is fairly similar to its predecessor, AsFix2, which is used by SDF2 and the SGLR and JSGLR parsers.

Generic Parse Tree Tokeniser The parse tree tokeniser (available in UPTR and AsFix2 versions) traverses a parse tree, and produces data tokens for leaf node; i.e., lexical (corresponding to identifies, numbers and such), literal (corresponding to keywords and punctuation) or layout (corresponding to white space and comments) nodes. The layout nodes are split into horizontal space (`WS`), newlines (`NL`) and comments (`COM`), and literals and lexicals are categorised according to a customisable scheme:

– the text of each token is checked against a category mapping table (one for lexicals and one for literals) either by exact string matching or regular expression matching;
– the parse tree node is checked for a category annotation (such annotations can be added automatically by a parser); and/or
– all tokens of the same syntactic sort can be mapped to the same category.

Remaining non-space tokens are categorised as the default `TXT`.

Additionally, control tokens may be emitted for some non-leaf nodes in the parse tree – for example, indentation may require `#BEGIN`/`#END` control tokens around lists of statements and other parts of the code that should be indented.

Mapping tables for literals, lexicals and sorts are all configurable, allowing the tokeniser to be customised to a particular language, and having it output tokens categorised according to a common scheme.

As an example, consider the following Java code input:

```
if(b) { x = 3; }
```

After parsing it using an SDF2 grammar[3], the AsFix2 tokeniser yields the following token stream:

```
"if":IF "(":LPAR "b":ID ")":RPAR " ":WS "{":LBRC #BEGIN " ":WS "x":ID
" ":WS "=":EQ " ":WS "3":NUM ";":SEMI " ":WS #END "}":RBRC
```

A further refinement would be to tokenise comments, so they can be reformatted as well.

2.3 Token Processors

A token processor accepts a stream of input tokens, and produces a stream of output tokens. Each processor has a `process` method which performs one step of the processor. The pipeline framework will ensure that `process` is called whenever there is available input to process.

The processor connects to the overall pipeline through a *pipe connector* which provides buffering (if needed), a history of recently outputted tokens (if needed) and may also provide transparent handling of control tokens for processors that are only interested in data tokens.

We are free to implement a `process` in any way, as long as it satisfies the general interface, but for convenience and performance, we provide a rule-based framework for specifying processors. This should be suitable for most simple

[3] JavaFront – `http://strategoxt.org/Stratego/JavaFront`

cases; more advanced processors can be programmed in a general purpose language. Section 2.4 and Section 2.5 give examples of token processors for adjusting spacing and breaking lines, respectively. Full versions of the processors and additional processors (including a general indenter) can be found in the online materials.

Rule-Based Token Processors A rule-base processor makes decisions based on a set of prioritised rules. Each rule consists of a condition and an action. In our Java library, conditions and actions are built using static methods returning condition and action objects. Two kinds of conditions are currently supported:

- `at(CAT...)` true if the next incoming token(s) match `CAT...` ("looking at")
- `after(CAT...)` true if the last emitted token(s) match `CAT...`

For example, the condition `after(A, B).at(C, D)` matches if the next tokens match category `C` and `D`, and the last emitted token was a `B` preceded by an `A`.[4] A token $\texttt{t:C}_t$ matches a category `C` iff $\texttt{C}_t = \texttt{C}$ or $\texttt{C}_t$ `<:` `C`.

An action may be arbitrary Java code, or selected from a simple library of actions:

- `insert("txt", CAT)` – insert a token before the current token
- `move` – move the current token from input to output
- `seq(a1, a2, ...)` – execute actions in sequence
- `drop` – delete the current token without producing output

Rules are added to the processor using the `addRule` method. For example, the following processing rule deletes all spaces:

```
addRule(at(SPC), drop);
```

We'll see more examples in Section 2.4.

Rule processing is implemented using decision tables, where a pair of categories are looked up, resulting in an action to be executed. We currently only support matching with two categories – either a `after(A).at(B)` pair, or a two token look-ahead, `at(A, B)`. If only one category is given in a rule, the other is assumed to be `TOKEN`, the supercategory of all categories.

Rule Priorities It will often be the case that there is more than one matching rule in a given situation. The general rule for resolving such ambiguities is that the more specific rule should apply. In this context, "more specific" means, for a rule with two categories:

- Assuming we are looking at a token with categories C, D, and we have two rules r_1 and r_2 with conditions involving C_1, D_1 and C_2, D_2 respectively, with C `<:` C_1, C `<:` C_2, D `<:` D_1, and D `<:` D_2;

[4] It is useful to think of the tokens as flowing in from the right (or, equivalently, that the token processor moves left-to-right over the token stream).

– rule r_1 is more specific than rule r_2 if $dist(C, C_1) + dist(D, D_2) < dist(C, C_2) + dist(D, D_2)$,
– where $dist(A, B)$ is the number of steps in the shortest supercategory chain A `<:` … `<:` B from A to B.

In addition to this priority rule we may also state that one set of rules should always have priority over another set of rules. When building a rule processor, this can be accomplished with the `addPriorityLevel` method.

Potential conflicts can be determined when the decision table is built, and a suitable warning will be provided to the programmer.

2.4 Token Processor: Spacer

The duty of the *spacer* is to process a stream of tokens, and insert spaces as appropriate. This is used to control horizontal spacing in a document.

Spacing can increase readability, even though many languages do not treat spaces as significant. A naive pretty-printing may insert spaces between all tokens, just to be on the safe side (e.g., preventing identifiers from running into each other). A more refined approach is to insert spaces according to style rules.

Spacing is readily implemented with a rule-based processor. Since many spacing rules are similar across languages (e.g., "`(x, y)`" is preferable to "`( x ,y)`"), we expect to be able to reuse much of the spacing code for multiple languages.

For a Java or a similar language, some sensible spacing rules might be:

– No spaces on the inner side of parentheses
– Always (or never) space between an `if` and the parenthesis
– Always space after a comma, never before
– No space before semicolon
– Always space around a binary operator
– Always spaces between any other tokens

We may implement these rules using the code in Fig. 3. The rule set removes all spaces from the input (highest priority level), and – unless otherwise specified – adds a new space between all text tokens (lowest priority level).

If we apply the rules to the following Java code:

```
if(b) {x=3;}else{x=4;}
if(b) x = f ( 1,2,3);
```

we get the following result, with some spaces inserted and some removed:

```
if (b) { x = 3; } else { x = 4; }
if (b) x = f(1, 2, 3);
```

A user-friendly configuration frontend could present customisation choices to the user, and then select appropriate rules to implement them. Note that it is also possible to specify certain rules to be followed (e.g., no space before comma, always after), while not otherwise changing the input.

```
addRule(at(WS), drop); // delete all incoming spaces

addPriorityLevel(); // rules above have highest priority

addRule(after(LPAR), nop); // no space after left parenthesis
addRule(at(PAR), nop); // no spaces before parentheses
addRule(after(IF).at(LPAR), space); // but always between 'if' and '('

addRule(at(PUNCT), nop); // no space before comma, semicolon, etc.

addRule(after(TXT).at(BINOP), space); // space around binary operators (this
addRule(after(BINOP).at(TXT), space); // actually follows from the general rule)

addPriorityLevel(); // rules below have lowest priority

addRule(after(TXT).at(TXT), space); // general rule
```

Fig. 3. A selection of sample spacing rules. The full spacing code for Java is available online. The command `space` inserts a space before the token we're looking `at`; `drop` deletes the token, and `nop` keeps the token (used for overriding a more general rule).

2.5 Token Processor: Line Breaker

The *line breaker* is responsible for turning a token stream into something that is ready for printing or converting into a string. This means reducing all formatting related stream content into nothing but text and line breaks. If line breaking is done, any indentation must be done together with it, since indentation has an impact on line width (though it is certainly possible to do only indentation in a pipeline, without line breaking).

Line breaking is easily the most challenging part of code formatting, involving trade-offs between horizontal and vertical space usage, code clarity and aesthetics. Also, earlier breaking decisions can impact later ones, in terms of how much space is available on a line – making the 'wrong' choice can make it impossible to achieve a pleasing result later.

We have experimented with three different line breakers. One is experimental, and based on assigning a 'breakability' value to each space (based, e.g. on how deeply nested an expression is); the higher its value, the more breakable it is, and the more likely it is that we break the line at that point, particularly as we get closer to the end of the line (Section 3.2). Another one is a refinement of an algorithm by Wadler [18]; some extensions have been added, and adjustments have been made to accept stream-based input in order to fit the pipeline (Section 3.3). The third one is a recent algorithm by Kiselyov et al. [9], with a novel implementation technique, reliant on stream-based processing (Section 3.4).

The experimental algorithm is implemented in Rascal only, and is not yet optimised for performance. The Wadler algorithm is implemented in both Rascal and Racket (with Racket offering decent performance). The Kiselyov algorithm

is implemented in Racket only, with a present shortcoming denying us the algorithm's theoretically pleasing performance characteristics. We have no clear favourite among these line breakers, but with our library of pluggable components we aim to avoid "lock-in" to any particular choice – and also to allow comparison of the different algorithms.

3 Line Breaking

3.1 Nesting and Indentation

Proper indentation relies on information about the current nesting level of the code. Additionally, one of our algorithms (Section 3.2) relies on nesting for making line-breaking decisions.

As we are working on a stream of data rather than a tree where nesting is explicit, we rely on control tokens to tell us about changes in the nesting level. The current nesting level can be tracked using a stack.

In addition to the usual increasing and decreasing of indentation levels, our nesting primitives supports absolute (specified column), relative (to the current column), and string-based indentation, as well as non-indenting nesting. Absolute indentation is useful for printing `#ifdef` and other CPP directives starting from the first column, or Lisp `;;;` comments. Relative indentation support makes it easier to align related text appearing over multiple lines. String-based indentation is necessary for pretty-printing "line comments" (e.g., comments starting with // in C++), if they are to be allowed to be broken over multiple lines.

The available indentation controls are:

- `LvInc(n)` – increase level by n (negative OK)
- `LvStr(s)` – append indentation string s
- `LvAbs(n)` – set new indentation level to n
- `LvRel(n)` – set level relative to current output cursor column

For example, a list of statements in a block may be surrounded by `#BEGIN(LvInc(1)) ... #END`, causing the statements to be indented one level more that the surrounding context. Relative indentation is useful inside parentheses. For example, the control tokens in the following stream would cause the line breaker to store the current column c, and indent the following line to $c + 0$ if the line is broken inside the parentheses:

```
... "(":LPAR #BEGIN(LvRel(0)) ... #END ")":RPAR ...
```

A plain `#BEGIN ... #END` is ignored by the indentation code, but may be used for other purposes, such as the algorithm in Section 3.2.

Our indentation control tokens are similar to Chitil's [2], who provides `OpenNest` and `CloseNest`. The difference is that `OpenNest` takes a function rather than an interpreted level value. This solution is more flexible than ours in that any function computing a new indentation level in terms of the current indentation level and "cursor" column is allowed. However, the function signature only permits integers, and hence the equivalent of `LvStr` is inexpressible.

3.2 A New Line-Breaking Algorithm

This line breaking algorithm tries to balance simplicity with the desire to keep related code on the same line. It is based on assigning breakability factors to spaces in the document (we'll assume that we always break lines at a space; if necessary, an empty space may be used as a break-point). The algorithm assumes that control tokens are inserted into the stream to indicate nesting, and the breakability is chosen based on this; the deeper the nesting, the less breakable a point is. Additionally, breakability can be set directly on space tokens, for "always break" and "never break". In our experiments, we've used nesting for each level of declarations, statements and expressions in our parse tree.

The Algorithm

- We keep track of
  - the desired line width (`W`),
  - the current breakability level (`B`) – a number between 0 and 1,
  - a stack of indentation levels (`I`),
  - and the current horizontal position, as a fraction of the line (`P`).
- Furthermore we have
  - A queue of processed tokens that have not yet been output
  - The last space that seemed like a good place to break (`S`)
  - The desirability of breaking at the last space (`D`) – computed based on `P` and `B` (based on experimentation, $P * B^2$ seems to be a good starting point). Initially zero, higher is better.
- For each incoming token, we do:
  1. - *if text*: append to queue
     - *if explicit line break*: flush queue, break line, indent, reset variables
     - *if nesting start*: decrease breakability by some factor (e.g., `nestFactor=0.75`)
     - *if nesting end*: increase breakability again
     - *if space*: we must decide whether this is a better place to break than our previous candidate. We compute the desirability of breaking at the current point, if it is higher or equal to `D`, we:
       (a) flush the queue
       (b) store this space as the best break point
       (c) store the new value of `D`
       Otherwise, the space is added to the queue.
  2. If the current position + queue length is larger than `W`, we:
     (a) break line, indent, reset variables
     (b) flush queue, reset best break point to empty, `D=0.0`

The algorithm never examines a token more than once, and has linear time complexity. In terms of memory, it needs a queue buffer proportional to the desired line width. Note that we only make the final decision on where to break when we reach the end of the line. Some tuning is necessary in order to find the best way to calculate how "desirable" a particular potential break is.

In the small example below, the algorithm has been applied to a code fragment with nested expressions, with line width 15. Compared to naive breaking nearest the end of line (right), our algorithm (left) tries to keep the deeper nested expressions on the same line; in this case, at the cost of using an extra line.

```
x = a * b
+ c / d
+ (c / d * f)
+ c / d;
```

```
x = a * b + c
/ d + (c / d
* f) + c / d;
```

3.3 Adaptation of Wadler's Pretty-Printing Algorithm

Our next line breaker is based on a pretty-printing algorithm described in a paper by Wadler [18]. We discuss the key concepts and characteristics of the original algorithm, and present our stream-adapted version and its extensions.

Wadler's Pretty-Printing Algorithm The original implementation [18] of the algorithm is Haskell based. The input given to the algorithm is specified as a *document*, which can be composed using provided operations. The most important operations include: `nil` (empty document), `text`, `line` (line break), `<>` (concatenation), `nest` (indented block), and `group`. The `group` constructor may produce a document whose layout involves line-breaking decisions, as the layout may differ depending on page width w.

The Haskell-based algorithm operates on primitive document types, which are: `NIL`, `:<>` (concatenation), `NEST`, `TEXT`, `LINE`, and `:<|>`. All except `:<|>` have a direct counterpart in the list of operations given above. The `:<|>` primitive signifies a *union* (or choice) of two possible sets of layouts, and any line-breaking sensitivity within documents (including those produced by `group`) must ultimately be expressible in terms of unions. The semantics of a union is that the left choice is taken if and only if it (fully, or up to any `LINE` break) fits on the line (i.e., the line width will not exceed w characters).

Use of the union primitive easily results in huge documents due to combinatorial explosion. For performance it is crucial for implementations of Wadler's algorithm to: (i) never inspect more than w characters per choice, and to (ii) not build (parts of) documents that are not inspected. Together these two measures achieve the property of *boundedness* [18] (of a pretty-printing algorithm), which Wadler defines as not looking at more than the next w characters in making line-breaking decisions. Measure (ii) is implementable by means of *lazy evaluation*, which one gets "for free" in Haskell as it is the default semantics. Laziness leads to a kind of co-routine computation [2], and it is also possible to encode sufficient laziness in strict languages [19].

Our Adaptation of the Algorithm Our layout algorithm is the same as Wadler's in the sense that the same primitive document types are supported. They may appear somewhat different, however, due to our requirement for a stream-based interface; `NIL`, for example, is simply represented as the empty

stream, whereas any two consecutive tokens in a stream can be thought to have been composed by `:<>`. Another difference to the original algorithm is that we chose to extend the `NEST` primitive to support the indentation controls of Section 3.1, for additional flexibility. The extension adds expressive power without taking anything away or affecting the general performance characteristics of the original algorithm.

We have implementations in both the Racket and Rascal languages. These bear little resemblance to the original Haskell-based implementations, both due to Haskell's different evaluation semantics (lazy vs strict) and the different "document model" (objects vs streams).

Complexity Bounding Measures In the original implementation Haskell's need-driven evaluation automatically takes care of the bounding measures required for good performance. Rascal and Racket are both strict languages, and we had to implement both of the measures explicitly.

To account for measure (i) we implemented the actual layout algorithm in a strict manner, making state and modifications to it explicit. All state is stored in structures, and no recursion is used by the algorithm proper. This makes the evaluation order and control flow clear, and no information enabling early pruning of fruitless search paths is "hiding" somewhere up the call stack. Decisions about whether or not to backtrack are made as soon the right margin is crossed when examining a choice (of a union). Backtracking is implemented by switching to an older, stored state, which is possible as state updates are purely functional.

Measure (ii) is more externally visible as documents are provided as input, and operations for constructing documents, whether lazily or otherwise, must be made available to the user. In many cases the required laziness can be hidden by accepting reasonably-sized (and complete) "instructions" for building documents as arguments to operations, and then performing the construction lazily within said operations as appropriate. The on-demand construction in a stream setting is enabled by defining a lazy (functional) stream type. Many of our document construction operations return such lazy streams, conforming to the "even" style of laziness [19].

While we do expose our `Union` primitive and the lazy stream API to allow for full document building flexibility, a variety of typical layouts can be achieved using the higher-level operations that we provide. These operations include `flatten`, `group`, and `fill`, as documented by Wadler [18]: `flatten` replaces each line break (and its associated indentation) by a single space; `group` adds a `flatten`ed form of a document as the preferred choice; and `fill` takes a list of documents and creates a fill layout for them, so that whenever there are two or more documents left, the first two are laid out `flatten`ed and a single space between them if they fit on the line, and otherwise the first document is laid out unflattened and followed by a line break.

3.4 Kiselyov et al's Pretty-Printing Algorithm

We have also made a Racket port of the linear-time, backtracking-free, bounded-latency pretty-printing algorithm by Kiselyov et al [9]. The algorithm is faster than Wadler's, and operates on a token stream, after any initial tokenisation. The set of supported formatting operations is like Oppen's [14]. The algorithm is particularly interesting for us as it suggests a potentially convenient way to organise a pipeline with interleaved operation of token processors, similar to our own formatting pipeline.

The Kiselyov algorithm itself has an internal token pipeline making use of a `yield` construct similar to what one finds in Ruby and many other languages [7]. Kiselyov's implementation of `yield` however does not require first-class delimited continuations (such as created by Racket's `call-with-continuation-prompt`), but rather is based on much lighter-weight *simple generators*. Their idea is to bind each token processor's `yield` target in the dynamic environment as a function. Our implementation of `yield` uses Racket's built-in support for dynamic binding (i.e., "parameterization"), which is semantically equivalent to the original Haskell implementation's use of the `Reader` (or "environment") monad.

Our initial impressions of using the simple generator style of co-routines for building pipelines are positive. One can just write "regular" Racket functions that may invoke `yield` to emit tokens, and the bindings can be established only once composing the pipeline. Processors requiring state may simply retain it in their closure.

The one problem with our current implementation is that one of its auxiliary data structures does not have all of the required stringent algorithmic properties. We are currently using an implementation of Okasaki's *banker's deque* [13], which does not enable concatenation and iteration with the required properties. The pipeline itself requires no explicit data structure, thanks to the `yield`-based approach of immediately passing tokens from one processor to another, without any pipe data structure in between.

4 Plumbing

4.1 Java Pipeline Design

We have so far described the details of token processors and formatting components, we'll now go into the details of the pipeline itself and how components are coupled together – the plumbing.

Our generic pipeline framework for Java is built on three concepts:

– *processors* that manipulate a stream of objects;
– *pipe components* that control the processors, and are connected together in a pipeline; and
– *pipe connectors* that connect processors to pipe components, and provide buffering and various utility services.

The framework is designed to support (optional) concurrency with each processor running in its own thread. The framework takes care of transporting the information through the pipeline, without the processor implementer having to worry about locking or concurrency issues (unless the processor accesses non-local information). Streams may be of any kind of object – in our PGF formatting framework the objects are tokens.

Pipe components supports a `connect` method for connecting the output to another component, `put` for sending an object into the component, `end` for signalling the end of a stream, and `restart` for resetting any internal state and preparing for a new stream of data. In a concurrent setting, the pipe component will take care of managing a processing thread.

Each pipe component connects to its processor through a pipe connector. The connector can provide input and output buffering, and can also maintain histories of the last seen or last output data items. The buffering is useful to provide look-ahead for the processor, and also to reduce the amount of locking in a concurrent setting. Multiple connectors may be layered to provide filtering or translation of the data stream (for example, processing control tokens).

Each processor has a `process` method, which is called by the pipe component whenever there is input available through the pipe connector. The processor may inspect or get objects from the input buffer, and also send objects to the output. The processor can state the amount of look-ahead and output history needed, and also how the start and end of the stream should be signalled.

Activity is driven by `put`ting objects into the pipeline; `put` calls to the first component will propagate calls through the entire pipeline; the output from one processor triggering a `put` to the next component which triggers the next processor, and so on.

4.2 Grouping and Dynamic Processors in Racket

The pretty printer adaptation in Section 3.3 is designed for pipelining, in the sense of contributing to a result as input becomes available. For example, the formatting of a nested range of tokens can proceed before the end marker of the range has been seen. The nesting construct is easy to handle in a stream setting as any tokens within a nested range may be processed unconditionally. This is unfortunately not the case with *union* primitive.

As mentioned in the subsection on complexity bounding measures, it is imperative for performance not to inspect (or even build) all the content of "large" unions. Hence such content should not appear within the stream in a "flat" form, as then it would be necessary to scan through any union contents just to get past them. We are forced to "unnaturally" express a union as a token containing the left and right choices as embedded streams. These nested streams can then be inspected to the appropriate extent.

Alternative and lazy constructions may not be natural within a token *sequence*, but often such constructions are just the low-level expression of what really is a concrete sequence with some associated semantics. For instance, consider the `group` function, defined in Racket as shown below, involving the `Union`

primitive and a lazy `flatten` operation. In the (commented out) example `group` expression, we essentially just have the word sequence of "a" followed by "b", with the semantics that the words should be laid out horizontally if they fit on a line, and vertically otherwise.

```
;; E.g. (group (tseq "a" br "b"))
(define (group ts) ;; tseq → tseq
  (Union (flatten ts) ts))
```

tseq is our token sequence datatype in Racket: `empty-tseq` is an empty sequence, `tseq` is a constructor, and `tseq-put` appends a token, with functional update.

As can be seen from the listing, implementing `group` as a function in terms of other operations is trivial when the full token stream `ts` to be grouped is available. We want to retain this ease of implementation, but would still like to allow `group`s to be emitted incrementally, token by token, as is fitting for a stream setting. Chitil, in his pretty printer [2], enables this specifically for `group` by supporting `Open` and `Close` tokens for delimiting the contents of a `group`. To a similar and more general effect we want to be able to "encode" the argument value for `group` as a sub-sequence of tokens, and then have the result of invoking `group` with that argument used as input for the actual layout algorithm.

To allow for such encoding, we introduce a *grouping* mechanism, which dynamically inserts a processor into the pipeline, along with a substream of tokens to be processed. While this kind of preprocessing happens to be particularly essential for Wadler's algorithm (unlike e.g. for Kiselyov's and our own), we have implemented the grouping processor in a fairly general way, and might find useful applications for it elsewhere within our token pipeline.

In a general sense the grouping facility makes it possible for "foreign language" (e.g., arguments to a function) to appear within an input stream, provided that it is known where the "foreign expression" begins and ends, and that a translator (to "native" language) is provided – for example, doing special formatting of comments. The former condition is met with opening and closing control tokens indicating the foreign range. The latter condition is met in an open-ended way by specifying a translator in the opening `Begin` token. A translator must conform to the `Grouping` abstraction, which includes functions for managing state for the delimited token range.

Below we define XML-inspired `group/` and `/group` delimiters, and use `tseq-put` to cache arguments for the `group` function. We then have `group` called with its entire cached argument upon seeing an `End` token. In this example the result is a single `Union` token, and hence there is no possibility of returning result tokens incrementally. The `Union` token may be lazily constructed, however. Here the specified `#:end` operation produces the full result (as a stream), which indeed is lazily constructed. The `#:put` operation is expected to receive and incorporate both new input as well as the results of any contained, nested groupings.

```
(define group-grouping ;; Grouping
  (make-grouping 'group
   #:new (thunk empty-tseq) ;; create fresh state
   #:put tseq-put ;; incorporate token into state
```

```
  #:end group)) ;; finalise by calling 'group' with state

;; E.g. (tseq group/ "a" br "b" /group)
(define group/ (Begin group-grouping))
(define /group (End group-grouping))
```

A grouping implemented in such a manner (i.e., call a function to get the result) may cause prohibitive space consumption for some applications, as depending on grouping semantics one may have to first buffer the entire input and then the entire result, and these can generally be of arbitrary length. However, depending on the grouping it may be possible to reduce space consumption by: (i) having "argument" tokens incorporated into a (small) result as they are read, without having to buffer them (cf. a hash function); and/or (ii) returning the result as a (small) lazy stream which will compute the full result sequence one token at a time (cf. a generator).

The `group` function gives us (ii), but (i) appears impossible, and we are still left with worst-case linear space consumption for our grouping implementation. We presently have no solution to avoid such overhead for all groupings, and to begin to achieve that we suspect that our grouping mechanism would have to be made less generic, more closely integrated with the algorithm proper, or both. As shown by Chitil and Swierstra [2, 17], achieving theoretically pleasing layout performance can be challenging and intricate.

The grouping support mechanism is implemented by maintaining grouping state within token sequences to ensure correct ordering as tokens flow in and out of grouping processing. Multiple groupings may nest, and groupings may output groupings; there is no guarantee of termination. No token reaches the layout algorithm proper before any grouping processing concerning it is complete; the algorithm proper is unaware of groupings.

5 Discussion

5.1 Plumbing Considerations

The concept of organising a code formatter as a pipeline is appealing. It offers four benefits, as far as we can tell:

- a clear separation of concerns,
- flexibility in that components can be plugged together to achieve different effects (e.g., indent only; spacing + indentation; line breaking + indentation),
- ease of prototyping and experimentation, since multiple interchangeable versions of one component can be tried,
- a possibility for fairly simple concurrent processing.

There are different ways to achieve pipelining. Our Java library uses a series of interconnected objects to achieve fairly straight-forward pipelining in Java. The technique of Kiselyov et al [9] uses the `yield` language construct to achieve "native" pipelining in languages that support it, without having to manage the

pipeline as a data structure. The grouping construct of Section 4.2 is designed for use in a surrounding pipeline, but also provides a way to dynamically build a more complex pipeline based on the control tokens in the incoming token stream.

Our Java version has advantages in the relative ease of doing buffered look-ahead and concurrency; a `yield`-based pipeline offers relative simplicity for languages that have it or where it can be added. Overall, we feel the choice comes down to the preferred (or required) implementation language.

5.2 Performance

Although performance may be of some importance in an interactive setting, it is of less concern than the quality of output, as long as "decent" performance is offered. In our experiments with our Java implementation of PGF, the most time consuming part of formatting is the tokenisation of parse tree – but even this is dominated by the time spend on parsing. Formatting a 2 MiB file of Java source takes around 6s (excluding parsing) on our workstations, which is a bit slower than Eclipse's Java formatter, but not much.

5.3 Quality and Reusability of Processors

We had an expectation that the various processing components that make up the formatting pipeline should be fairly easy to adapt to different languages, provided that the tokenisation has been customised for the particular languages. Our results so far are promising; generic spacing, indentation and line-breaking processors do a fair (but by no means perfect) job of formatting Java and Magnolia code. Producing high quality output requires careful tuning, and will of course also have to take into account styling preferences. Examples of output can be found in the online materials.

More work is needed on composing and combining rule sets of rule-based processors; extending a generic processor with rules for a particular language has a tendency to result in rule conflicts or unexpected behaviour.

5.4 Related Work

Oppen's pretty printer [14] is also stream oriented. Unlike Wadler's, it exploits streams to allow two communicating sequential processes to coordinate their work. This results in performance characteristics of time $\mathcal{O}(n)$ and space $\mathcal{O}(w)$, with input length n and page width w. In comparison, the worst case time complexity of Wadler's algorithm is $\mathcal{O}(nw)$ [2], i.e. superlinear if w is considered variable.

While Oppen's solution is unbeaten in performance, it is also monolithic. The constructs of his layout language only exist as part of a whole, without a meaningful description in isolation [5]. The variety of parameters for Oppen's *blank* tokens add expressiveness to the language. For example, *consistent* and *inconsistent* blanks loosely correspond to `group` and `fill`, respectively. The *variable*

offset of blanks has the same motivation as the `LvRel` parameter we introduced in Section 3.1.

Hughes' pretty printer [5] is based on algebraic design, and served as a major influence for Wadler's. The `Union` operator, for instance, has its origins in Hughes' algebra, and indeed in the set theoretical $\cup$ operator. Hughes' layout operator set is expressive, able to express some layouts that Wadler's cannot [18], but also such that a bounded implementation is impossible [18].

Chitil [2] managed to devise a purely functional pretty printer with Oppen-level efficiency. In it he also opts to represent documents as token sequences rather than trees, due to sequences appearing more amenable to efficient processing.[5] Chitil maintains context information in explicit data structures, and notes that this is important for achieving high efficiency. Our implementation of Wadler's algorithm shares these two design choices, though for reasons of architecture and clarity.

Swierstra's and Chitil's joint work on a pretty printer [17] resulted in two linear-time implementations that are simpler and clearer than Chitil's earlier work. They provide the insight that such solutions can be achieved by utilising two mutually recursive processes running asynchronously, but that such solutions are hard to express in purely algebraic style. Wadler's algorithm was derived based on algebraic techniques.

Kiselyov et al. found a still simpler and clearer way to implement a linear-time, bounded-latency solution to the pretty printing problem [9]. Their algorithm, discussed in Section 3.4, is similar to Swierstra and Chitil's second solution in that instead of relying on Haskell's lazy evaluation to interleave processing, a co-routine style approach is used; Swierstra and Chitil's approach is more heavyweight in that it involves building and storing "continuation" functions. Kiselyov et al. point out a sometimes overlooked [16] corner case in the processing of `groups`, offering normalisation of document trees as the solution. We believe such normalisation cannot be done efficiently on token streams, meaning that bounded look-ahead is lost unless we can ensure that earlier token processors only produce normalised output.

The Box [1, 3, 8] formatting model is based on composing two-dimensional boxes of code. This produces good quality output, but the complexity of the algorithm is high, leading to poor performance on large documents (often dominating the other processing steps in a source-to-source transformation). Additionally, some forms of indentation are difficult to express.

Jackson et.al. [6] provide an efficient, stable peephole pretty-printing algorithm, suitable for use in an interactive editor. The peephole property means that it is capable of pretty-printing just a part of a program, corresponding to an editor view. The running time of the pretty-printer is thus independent of the full length of the program—only the size of the editor's view (or peephole) matters. The algorithm is *stable*, in that it gives the same result as if the entire program was pretty-printed, avoiding reformatting artefacts as the user

[5] Swierstra has shown that a tree representation does not preclude Oppen-level efficiency [16].

scrolls the editor view. The stable peephole is achieved by identifying *anchors* in the program; places where the indentation level is the same, no matter which choices are made by the formatter. The line-breaking algorithm itself is a variant of Wadler's [18].

Our new line breaker has (we believe) fewer choices than the Wadler algorithm, so it should be easier to identify anchors, and provide stable peephole functionality. However, peepholing may not work so well with a pluggable architecture. We need to explore this more.

Reiss [15] has shown that given samples of existing source code in the desired style, machine learning can be used to deduce formatting rules for the style. This approach should also apply to deducing rule tables for our spacer engine, for example.

6 Conclusion

PGF is a general framework for code formatting, based on a flexible pipeline of formatting components. We aim to use the framework as a basis for further experimentation into customisable, language-independent formatting components. The flexibility of offered by pluggable components seems useful for conducting such experiments. We provide both a rule system for implementing components, and support for using general purpose languages.

We have built components for spacing, indentation and three variants of line breaking, as well as tokenisers accepting parse trees in UTPR and AsFix2 format. Although we are implementing PGF as a Java library, we also perform experiments and prototyping in Rascal and Racket. This has allowed us to rapidly try out various techniques, including three different line breaking algorithms.

The general pipelining framework should be reusable for other purposes than just code formatting – it is built to be independent of the type of data processed. We have applied the PGF formatter to the Java and Magnolia languages, so far with promising results, though further tuning is needed to produce high quality output, and to determine the level of code reuse possible when implementing formatters for multiple languages.

Online materials are available at `http://nuthatchery.org/sle13/`

Acknowledgements Thanks to Eivind Jahren, who has helped us understand unfamiliar Haskell concepts. This research has been funded by the Research Council of Norway.

References

1. van den Brand, M.G.J., Visser, E.: Generation of formatters for context-free languages. ACM Transactions on Software Engineering and Methodology 5(1), 1–41 (January 1996)

2. Chitil, O.: Pretty printing with lazy dequeues. ACM Trans. Program. Lang. Syst. 27, 163–184 (January 2005)
3. Coutaz, J.: A layout abstraction for user-system interface. SIGCHI Bull. 16(3), 18–24 (Jan 1985), http://doi.acm.org/10.1145/1044201.1044202
4. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. (2010), http://racket-lang.org/tr1/
5. Hughes, J.: The design of a pretty-printing library. In: Advanced Functional Programming. pp. 53–96. Springer-Verlag (1995)
6. Jackson, S., Devanbu, P., Ma, K.L.: Stable, flexible, peephole pretty-printing. Science of Computer Programming 72(1-2), 40 – 51 (2008)
7. James, R.P., Sabry, A.: Yield: Mainstream delimited continuations. In: First International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011) (May 2011)
8. de Jonge, M.: A pretty-printer for every occasion. In: Ferguson, I., Gray, J., Scott, L. (eds.) Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000). pp. 68–77. University of Wollongong, Australia (Jun 2000)
9. Kiselyov, O., Peyton-Jones, S., Sabry, A.: Lazy v. yield: Incremental, linear pretty-printing. In: 10th Asian Symposium on Programming Languages and Systems (APLAS). pp. 190–206 (Dec 2012)
10. Klint, P., van der Storm, T., Vinju, J.: Rascal: A domain specific language for source code analysis and manipulation. In: SCAM '09: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 168–177. IEEE Computer Society, Washington, DC, USA (2009)
11. McKeeman, W.M.: Algorithm 268: Algol 60 reference language editor. Commun. ACM 8(11), 667–668 (Nov 1965)
12. Miara, R.J., Musselman, J.A., Navarro, J.A., Shneiderman, B.: Program indentation and comprehensibility. Commun. ACM 26(11), 861–867 (Nov 1983)
13. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1999)
14. Oppen, D.C.: Prettyprinting. ACM Trans. Program. Lang. Syst. 2, 465–483 (Oct 1980)
15. Reiss, S.P.: Automatic code stylizing. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 74–83. Atlanta, Georgia (Nov 2007)
16. Swierstra, S.D.: Linear, online, functional pretty printing (corrected and extended version). Tech. Rep. UU-CS-2004-025a, Department of Information and Computing Sciences, Utrecht University (2004)
17. Swierstra, S.D., Chitil, O.: Linear, bounded, functional pretty-printing. Journal of Functional Programming 19(1), 1–16 (2009)
18. Wadler, P.: A prettier printer. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming. Cornerstones of Computing, Palgrave Macmillan (Jun 2005)
19. Wadler, P., Taha, W., Macqueen, D.: How to add laziness to a strict language without even being odd. In: Workshop on Standard ML. Baltimore, Maryland (Sep 1998)