A Generic Specification of Prettyprinters

Richard B. Kieburtz

Oregon Graduate Institute
19600 NW von Neumann Dr.
Beaverton, OR 97006 USA

Report. No. CSE-91-020

November, 1991, revised, February, 1992

Abstract

A prettyprinter interprets terms in the abstract syntax of a formal language as text, formatted for visual display on a screen or a printed page according to a set of rules for preferred layout and subject to constraints such as the dimensions of a page. Here we develop a generic, language independent, policy-driven specification for prettyprinters. It is actually a meta-specification, for we specify the interpretation, in the style of a denotational semantics, of a pattern-oriented, layout specification language. With this pattern language, a display designer can readily specify rules and policies by which to display the sentences of a particular term language. The paper concludes by validating a few important properties of our specification.

A Generic Specification of Prettyprinters

Richard B. Kieburtz

Oregon Graduate Institute
19600 NW von Neumann Dr.
Beaverton, OR 97006 USA

1. Generic specifications and software design

This report illustrates a specification technique that emphasizes the separation of policy from mechanism—a well-known principle of design for systems software. The application is to the specification of prettyprinters, which are formatting programs for term-structured data. This problem can be generalized along several dimensions. The particular abstract syntax to be printed can be made a parameter and the format for display another. However, the design of a function to effect a display from a term and a format specification can also be given as a composition of generic functions parameterized by specific policy functions. A generic specification is in part, a meta-specification of such a composition.

First, we shall specify that the language of terms to be displayed will be described by a context-free grammar for the term syntax. The term grammar becomes a parameter of a generic prettyprinter. Second, the format in which terms are to be displayed will be specified in a layout language whose terms consist of multi-line patterns. We specify a syntax for these patterns. A layout specification, given in this layout language, becomes a second parameter for a prettyprinter. Our meta-specification consists in giving semantics to the pattern-oriented display language. We choose to give the semantics in a mathematical framework of *categories* whose objects can be interpreted as datatypes and whose morphisms are functions. This kind of semantic specification can be directly realized as a program.

Even when specifying the semantics of display, there are opportunities for generality in the specification. We employ a new technique of composing semantic domains, based upon the notion of monad composition in a suitable category. It allows us to structure a specification so that its functionality can be developed incrementally. When state is introduced, it is partitioned into distinct components, which aids reasoning about a stateful specification.

A semantics specification can be sketched in outline form. Details are superimposed on the outline by particular policy functions that are given as its parameters. The outline determines the types of the semantic functions and also their control structure, which is induced by the types. The policy functions determine the detailed behavior. Policy functions are often very simple, and are the focal points for modifying a generic specification to tailor it to a specific application.

The abstract design of a software system is captured in a semantics specification, whatever may be its ultimate implementation. We favor deriving that implementation by automatic program generation and machine-supported transformation steps. It is because automatic generation is feasible that we consider the specification to be authentic design. A generated implementation necessarily inherits the functional properties verified of the specification.

There is a final point to keep in mind while studying the specification that follows. There is enormous leverage for design reuse when a design (1) is generic and can be used to produce many differently specialized instances; (2) is modular, composed of simpler, less specialized parts; and (3) provides a template for automatic generation of multiple implementations that may differ individually as to programming language, choices of data representations or the platforms on which they run.

2. Prettyprinting

Prettyprinting refers to the task of displaying the sentences of a formal language in an attractive format while respecting constraints on the layout, such as those imposed by the physical dimensions of a page. The problem is open-ended with regard to generality, for one may envision two-dimensional displays such as are commonly used in mathematics, the inclusion of diagrams, enclosing boxes, etc. We have limited our goals to tabular displays, i.e. to line-oriented displays with whitespace indentations from a uniform left margin, because this form of display is most commonly used for programming languages and because it presents sufficient challenge for a first attempt.

The inputs to a prettyprinter are:

(1) a term to be printed, structured by the abstract syntax of a formal language,

(2) a font mapping for operator symbols and identifiers occurring in the term (not all identifiers need to be mapped to the same font)

(3) the maximum scroll width for output pages.

The primary input is taken to be a term, rather than a list of print items, for several reasons: (i) layout policies will depend on the term structure, which must somehow be encoded if it is not manifest; (ii) a printer for a programming language should be capable of displaying synthesized terms as well as terms gotten by parsing a sentence originally input in concrete syntax by an author; (iii) it is often desired to have more than one display format for terms, and choices of concrete syntax, punctuation marks, keywords, bracketing, etc. may vary from one to another.

An advantage of a pattern-oriented specification language for layouts is that a display designer can more easily relate the formal specification, a pattern, to his/her intuition than is possible if the layout specification were procedural or utilized regular expressions. A disadvantage is that it is often more verbose to give explicit patterns for the display of each operator of the abstract syntax. We believe that a specification language must first serve as a vehicle to communicate concepts among humans, and have chosen a pattern-oriented language for its readability.

In formulating this specification, we have drawn on the experiences reported by others in specifying prettyprinters. The best-known, early attempt to give a systematic, language-independent specification was by Derek Oppen [4]. He defined a hierarchical organization of nested blocks of atomic print items. Each block is to be laid out as a unit. Blocks can be marked for display vertically, horizontally, or semi-horizontally, with line breaks inserted as needed to meet page constraints. His prettyprinter accommodates indentation offsets and matching of bracket pairs. It is language independent, but the prettyprinter algorithm is procedurally defined, rather than derived from a specification. Oppen's algorithm is optimized to produce the fewest number of line breaks, consistent with the page width and mandated indentation requirements.

More recently, John Hughes [1] has given a functional specification of language independent prettyprinting in terms of a set of five combinators. These combinators act upon the atomic print items or upon blocks of print items to produce vertical or horizontal layout with line breaks inserted to meet page width constraints. The separator combinator can introduce indentation following a line break. Indentation offsets are propagated by a combinator for nesting layouts. Text is defined in terms of a datatype *Idoc* (Intelligent document) which depends upon boolean-valued state variables that indicate the current layout mode (horizontal or vertical) and whether layout of the current block is to be restricted to a single line.

A layout specification, using Hughes' formalism, is procedural. For each form of term of the abstract syntax, the display designer gives a layout directive as a combinator expression. The designer can exercise somewhat more control over layout policy than with Oppen's fixed algorithm, although some choices are still preempted by the particular choice of combinators.

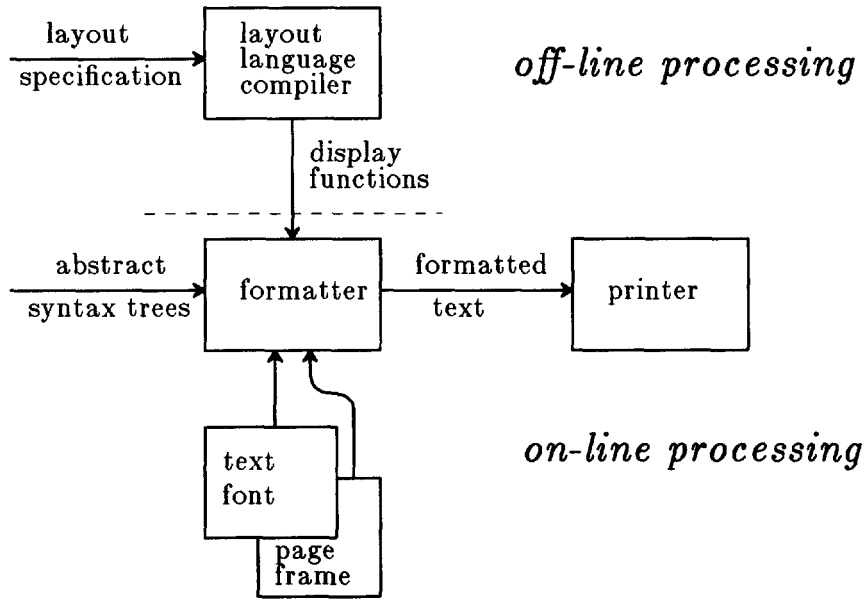

Figure 1: Software Architecture of a Prettyprinter

The formatting policy is presented as a sentence in a layout language, which is compiled to produce a formatter for the specified abstract syntax. Translation to a text font introduces attributes of width and height for each atomic text string. Page constraints may include field width and page height. The formatted text that is output will include inserted whitespace, line breaks and page breaks.

Both Oppen's and Hughes' prettyprinters employ backtracking. The backtracking in Oppen's is relatively restricted; each line of display is committed incrementally, thus mitigating the storage requirements for processed but uncommitted text. Hughes' algorithm permits a greater scope for backtracking in order to obtain the most desirable among feasible layouts. It probably minimizes the number of line breaks consistent with the specified layout policy, though this has not been proven.

One of our goals in proposing the specification given here has been to separate policy from mechanism insofar as possible. Thus, the combinators we have defined are parameterized on policy functions that may be predicates or state transformation functions. A layout designer who wishes to experiment with different policies than the ones we have proposed can see exactly where to make changes. Policy substitution will not invalidate the proofs of properties of our specification, so long as the stated constraints on policy functions are adhered to.

3. A language for specifying layout for a prettyprinter

The layout we envision is line-oriented. Each line may have a different height, calculated from the heights of its component icons. Lines do not overlap in a layout. Lines may be indented; control of indentation is one of the functions of the text formatter and must be

specified in the layout language. The horizontal extent of a line may be constrained by a field width constraint, to be honored by the formatter.

In the layout language, a prettyprinter specification associates a list of zero or more layout patterns with each operator of the abstract syntax. The patterns are listed in order of preference, on the assumption that horizontal linewidth constraints can be met. Thus, consequent patterns will ordinarily display more vertical structure than their predecessors. Indentation given in the patterns is tabular, in case the print font has variable character widths.

A layout specification for an operator of the abstract syntax may also include optional bracket symbols and a conflicts relation that will be tested to determine whether or not bracketing is required. Normally, the conflicts relation will specify operator precedence conflicts in the concrete syntax.

Each layout pattern can also be tagged with an integer 'stiffness' parameter that will be used to select among possible candidates for optional layout patterns, in case the horizontal linewidth constraint is not met by the default layout.

Symbols:

$<string>	abstract operator ID
α	term variable
α*	term list variable
<string>	literal

in which α designates a single alphabetic character. Leading whitespace and newlines are significant.

Syntax of the layout language

SPEC	::=	LAYOUT-LIST
LAYOUT	::=	HEAD OPTIONS
HEAD	::=	OP-ID (ARG-LIST) [CONFLICTS]
ARG	::=	LETTER
	|	LETTER *
CONFLICTS	::=	conflicts {(α-LIST) '[' OPERATOR-ID-LIST ']' STRING STRING}
OPTIONS	::=	==> (NUMERAL) PATTERN OPTIONS
	|	==> PATTERN

Syntax of the pattern language

Productions

PATTERN	::=	PATTERN-ELEMENT
	|	PATTERN-ELEMENT newline PATTERN
PATTERN-ELEMENT	::=	ATOM
	|	ATOM SEPARATOR PATTERN-ELEMENT
	|	α '{' SEPARATOR '}' *
	|	α STRING '{' SEPARATOR '}' *
	|	STRING α STRING

3.1. Examples

```
$*(x,y)
conflicts (x,y) [$+ $- $~ $IF] ( )
==>(0)
x\*y
==>
x
\*y


$IF(x,y,z)
conflicts (x,y) [$IF] ( )
==>(0)
if x then y else z
==>(0)
if x then y
else z
==>(0)
if x
   then y
   else z
==>
if   x
then
   y
else
   z


$BLOCK(s*)
==>(0)
begin s{; }* end
==>
begin
   s{;
   }*
end


$AP(f a*)
conflicts (f) [$IF] ( )
          (a) [$* $/$+ $- $~ $PR $AP $IF] ( )
==>
f a{ {
  }}*
```

4. Specification of formatting functions

4.1. Naive formatting functions

We shall propose a set of combinators with which to give semantics to a layout language. The combinators are morphisms of a bi-cartesian, closed category which has strong monads of state transformers and exceptions.

The basic idea is that a pretty-printer should transform abstract syntax to text. We could also extend pretty-printing to lists of AST's using the function map^{List}.

$$\begin{aligned} &\mathit{display} : \mathrm{AST} \rightarrow \mathrm{Text} \\ &\mathit{map}^{List}\ \mathit{display} : \mathit{List}(\mathrm{AST}) \rightarrow \mathit{List}(\mathrm{Text}) \end{aligned}$$

However, a prettyprinter may also insert literal strings (keywords, brackets, punctuation) into the output text. Thus there should be a function

$$\mathit{literal} : \mathrm{String} \rightarrow \mathrm{Text}$$

as well. A format pattern may specify an intermixture of term variables with literal strings. Thus we cannot simply map a single function over a list of AST's to obtain properly formatted output. Nor can the naive functions *display* and *literal* be composed. The naive approach is not sufficient.

4.2. Composable formatting functions

Let's try to invent formatting functions that take a list of AST's as an argument. First, however, let's review the mathematical structure called a monad. We are interested in the interpretation of monads as abstract datatypes. We characterize a monad by specifying a type mapping, or datatype constructor $\mathrm{T} : \mathit{Type} \rightarrow \mathit{Type}$, its *unit*, a polymorphic function $\eta^{\mathrm{T}}_{\mathrm{X}} : \mathrm{X} \rightarrow \mathrm{T\,X}$ that injects values from an argument type into values in the datatype, and the *natural extension* mapping for the monad, which 'lifts' a function $f : \mathrm{X} \rightarrow \mathrm{T\,Y}$ to a function $f^{*\mathrm{T}} : \mathrm{T\,X} \rightarrow \mathrm{T\,Y}$. These functions cannot be arbitrary; they are related by a set of equations (the so-called monad laws for the Kleisli-triple representation of a monad [2]):

$$(\eta^{\mathrm{T}}_{\mathrm{X}})^{*\mathrm{T}} = \mathit{id}_{\mathrm{T\,X}}$$

$$f^{*\mathrm{T}} \circ \eta^{\mathrm{T}}_{\mathrm{X}} = f$$

$$(g \circ f)^{*\mathrm{T}} = g \circ f^{*\mathrm{T}}$$

in which the equations must be well-typed, although we have omitted the typing annotations on all but the first equation. When it is clear from the context what type constructor is being discussed, we also omit the subscript T from the star superscript.

For any monad, one also has the polymorphic map function for that monad, given in terms of the unit and the star-extension:

$$\begin{aligned} &\mathrm{map}^{\mathrm{T}} : (\mathrm{X} \rightarrow \mathrm{Y}) \rightarrow \mathrm{T\,X} \rightarrow \mathrm{T\,Y} \\ &\mathrm{map}^{\mathrm{T}}\, f = (\eta^{\mathrm{T}}_{\mathrm{Y}} \circ f)^{*} \end{aligned}$$

One useful class of monads is called state-transformers [6]. The characterization of a state-transformer with state type S is:

$$\mathrm{St_S\,X} = \mathrm{S} \rightarrow \mathrm{X} \times \mathrm{S}$$

$$\eta_{\mathrm{X}}^{\mathrm{St_S}} = \lambda x{:}\mathrm{X}.\, \lambda s{:}\mathrm{S}.\, x, s$$

$$f^* = \lambda t{:}\mathrm{St_S X}.\, \lambda s{:}\mathrm{S}.\, \mathrm{let} x, s' = t\ s\ \mathrm{in}\ f\ x\ s' \quad \text{where } f : \mathrm{X} \rightarrow \mathrm{St\,Y} \text{ and } f^* : \mathrm{St\,X} \rightarrow \mathrm{St\,Y}$$

We make use of the monad of state transformers with state type $List(\mathrm{AST})$ to extend the naive formatting functions proposed above, defining instead the composable formatting functions

$$display' : \mathrm{AST} \rightarrow \mathrm{St}_{List(\mathrm{AST})}\,\mathrm{Text}$$
$$display' = \eta_{\mathrm{Text}}^{\mathrm{St}_{List(\mathrm{AST})}} \circ display$$

$$literal' : \mathrm{String} \rightarrow \mathrm{St}_{List(\mathrm{AST})}\,\mathrm{Text}$$
$$literal' = \eta_{\mathrm{Text}}^{\mathrm{St}_{List(\mathrm{AST})}} \circ literal$$

When either of these functions is applied to an argument of the appropriate type, it yields a functional value of the common type $\mathrm{St}_{List(\mathrm{AST})}\,\mathrm{Text} = List(\mathrm{AST}) \rightarrow \mathrm{Text} \times List(\mathrm{AST})$. Such values can be composed. The composition is the **symmetric tensorial strength** for the monad of state transformers,

$$\tau^{\mathrm{St}} : \mathrm{St_S\,X} \times \mathrm{St_S\,Y} \rightarrow \mathrm{St_S}(\mathrm{X} \times \mathrm{Y})$$
$$\tau^{\mathrm{St}}(f, g) = \lambda s{:}\mathrm{S}.\, \mathrm{let}\ x, s' = f\ s\ \mathrm{in}\ \mathrm{let}\ y, s'' = g\ s'\ \mathrm{in}\ (x, y), s''$$

Define the infix composition operator $(f\,;g) = (@ \times id_{List(\mathrm{AST})}) \circ \tau^{\mathrm{St}}(f, g)$, where $@ : \mathrm{Text} \times \mathrm{Text} \rightarrow \mathrm{Text}$ is the catenation operator. This composition is an instance of a generic function that is defined for any monad T that has a symmetric tensorial strength τ^{T}. The generic function is:

$$\varsigma^{\mathrm{T}} : (\mathrm{X} \times \mathrm{Y} \rightarrow \mathrm{Y}) \rightarrow \mathrm{TX} \times \mathrm{TY} \rightarrow \mathrm{TY}$$
$$\varsigma^{\mathrm{T}} f = \mathrm{map}^{\mathrm{T}} f \circ \tau^{\mathrm{T}}$$

and $(\,;) = \varsigma^{\mathrm{T}}(@)$. With the use of $(\,;)$ we can form compositions such as:

$$(display'\,x)\,;(literal'\,\text{"}foo\text{"}) : \mathrm{St}_{List(\mathrm{AST})}\,\mathrm{Text}$$

Although this composite has the desired type, it still contains a free variable of type AST. If the function $display'$ is replaced by

$$display^{List} : \mathrm{St}_{List(\mathrm{AST})}\,\mathrm{Text}$$
$$display^{List}\ [\,] = \epsilon, [\,] \qquad \text{where } \epsilon \text{ is the empty Text value}$$
$$display^{List}\,(x :: xs) = display'\,x\ xs$$

then we can form the composition

$$display^{List}\,;(literal'"foo") : \mathrm{St}_{List(\mathrm{AST})}\mathrm{Text}$$

which could be used to express the semantics of a format pattern 'X,"*foo*"'.

4.3. Attribute-controlled formatting

Formats may be governed by attributes, such as the indentation offset, that specify the placement of a text item in a line. Attributes can usefully be grouped into two classes: *Inherited* attributes may provide parameters to layout policy but are not necessarily cumulative with layout; *Synthetic* attributes are cumulative. When layout policy does not depend upon a synthetic attribute, it can appear solely as a component of the result of the layout function. More complicated is the case of an attribute that is both inherited and synthetic, such as the horizontal location of a text item in a line. For such an attribute we need a new state component, which can be introduced with an additional use of the state-transformer monad. The composition of two state-transformer monads yields another monad,

$$\mathrm{St}^2\mathrm{X} = List(\mathrm{AST}) \rightarrow \mathrm{Loc} \rightarrow \mathrm{X} \times List(\mathrm{AST}) \times \mathrm{Loc}$$

$$\eta_{\mathrm{X}}^{\mathrm{St}^2} = \lambda x.\ \lambda xs.\ \lambda y.\ x,xs,y$$

$$f^* = \lambda t.\ \lambda xs.\ \lambda y.\ \mathrm{let}\ x,xs',y' = t\ xs\ y\ \mathrm{in}\ f\ x\ xs'\ y'$$

where $f : \mathrm{X} \rightarrow \mathrm{St}^2\mathrm{Y}$

Attributes that are only inherited can be introduced by a state reader monad, which also has a symmetric strength,

$$\mathrm{Rd}\,\mathrm{X} = \mathrm{S} \rightarrow \mathrm{X}$$

$$\eta_{\mathrm{X}}^{\mathrm{Rd}} = \lambda x.\ \lambda \rho.\ x$$

$$f^* = \lambda t.\ \lambda x.\ \lambda \rho.\ \mathrm{let}\ x' = t\ x\ \rho\ \mathrm{in}\ f\ x'\ \rho$$

where $f : \mathrm{X} \rightarrow \mathrm{Rd}\,\mathrm{Y}$

$$\tau^{\mathrm{Rd}}(f,g) = \lambda(x,y).\ \lambda \rho.\ (f\ x\ \rho, g\ y\ \rho)$$

A composition of state transformer monads can also be composed with the state reader monad. The attribute-controlled formatting functions will have the types

$$display^{\mathrm{St}^2}\colon List(\mathrm{AST}) \rightarrow \mathrm{Loc} \rightarrow \mathrm{Attr} \rightarrow \mathrm{Text} \times List(\mathrm{AST}) \times \mathrm{Loc}$$

$$literal^{\mathrm{St}^2}\colon \mathrm{String} \rightarrow List(\mathrm{AST}) \rightarrow \mathrm{Loc} \rightarrow \mathrm{Attr} \rightarrow \mathrm{Text} \times List(\mathrm{AST}) \times \mathrm{Loc}$$

where Loc is a type of numbers and Attr is a collection of inherited attributes.

The inherited attributes that we consider necessary are:

ρ_1: Number	indentation
ρ_2: Number	stiffness
ρ_3: List(Token)×(String×String)	conflicts set—brackets

By convention, a pattern with a stiffness level of zero is intended to fail unless it can be printed on a single line.

The most basic layout function positions a literal string given as its first argument, and does not depend on the purely inherited attributes. We indicate by subscripts that this function is parameterized with respect to a type font, **f**, and two policy functions, ϕ and u. We also determine at this point that Text is a list type. The definition of $literal^{\mathrm{St}^2}$ is:

$$literal^{St^2}_{\mathbf{f},\phi,u}\ str\ xs\ y\ \rho = [\phi_{\mathbf{f}}(str)], xs, u_{\mathbf{f}}(y,str)$$

where $\phi_{\mathbf{f}}$: *String* $\rightarrow$ *String'* is a font mapping function for font **f**. The purpose of u: Loc$\times$String $\rightarrow$ Loc is to update the position attribute that marks the right end of the displayed text. We require of u that it have an *additive* property:

(*) $$u_{\mathbf{f}}(y, s_1@s_2) = u_{\mathbf{f}}(u_{\mathbf{f}}(y,s_1), s_2)$$

Choosing Loc to be a numeric type, let

(1) $$u_{\mathbf{f}}(y,str) = y + length_{\mathbf{f}}\, str,$$

where $length_{\mathbf{f}}$ measures the length of a string as rendered in font **f**.

4.4. Optimal layout control under constraints

Prettyprinting may be subject to layout constraints such as that the length of a line must not exceed the field width between margins on a page. Additionally, there is usually some optimality criterion for printing. We want to produce the layout nearest to optimal which also meets the constraints. We shall provide for backtracking, as a simple control mechanism that enables a prettyprinter to realize optimal or nearly optimal layouts. This will allow a format specification to be given as a sequence of patterns, monotonically decreasing in optimality but increasing in complexity so as to satisfy ever more stringent constraint parameters. When an attempt to lay out a text according to an earlier pattern fails to meet the constraint, the next pattern in sequence will automatically be tried. This strategy is often adequate to satisfy linear constraints.

For this purpose we further enrich our layout functions with exceptions. The monad of exceptions is ExX = X+E, where E represents a set of exceptions, here taken to be a singleton set. Let

$$\mathrm{T}\,\mathrm{X} = List(\mathrm{AST}) \rightarrow \mathrm{Loc} \rightarrow \mathrm{Attr} \rightarrow (\mathrm{X} \times List(\mathrm{AST}) \times \mathrm{Loc}) + \mathrm{E}.$$

The enriched formatting functions are typed as:

$$display^{\mathrm{E}}_{isOK,\mathbf{f},\phi,u} : \mathrm{T(Text)}$$

$$literal^{\mathrm{E}}_{isOK,\mathbf{f},\phi,u} : \mathrm{String} \rightarrow \mathrm{T(Text)}$$

$$literal^{\mathrm{E}}_{isOK,\mathbf{f},\phi,u}\ str\ xs\ y\ \rho = \text{if } isOK\ str\ y\ \rho \text{ then } [\phi_{\mathbf{f}}(str)], xs, u_{\mathbf{f}}(y,str) \text{ else FAIL}$$

When we omit the subscript on the functions $literal^{\mathrm{E}}$ and $layout^{\mathrm{E}}$, we mean the functions that use the definitions of $isOK$, ϕ and u given above.

In the definition above, the policy predicate $isOK$, given a string and current state, determines whether or not the line width constraint is satisfied. We require of this test that it satisfy a monotonicity condition, with respect to substrings, namely that

(**) $$isOK\,(s_1@s_2)\,y\,\rho = true \Rightarrow isOK\ s_1\ y\ \rho = true \text{ andalso } isOK\ s_2\ y\ \rho = true$$

A useful choice for this policy function is

(2) $$isOK\ s\ y\ \rho = u_{\mathbf{f}}(y,s) \leq scroll\_width \text{ orelse } \rho_2 = \infty$$

where y is the current horizontal position and the test $\rho_2 = \infty$ checks the stiffness parameter to see whether the policy forbids backtracking to an alternative layout.

4.5. Alternative layouts

When an attempt to display a term fails to meet a layout constraint, an alternative layout scheme may be available in the list of layout options specified for a production of the abstract syntax. We specify the composition of alternatives by an infix operator '$?$'.

The operator $?^{\mathrm{Ex}}\colon \mathrm{Ex}\,\mathrm{X}\times\mathrm{Ex}\,\mathrm{X}\to\mathrm{Ex}\,\mathrm{X}$ designates an expression with an exception handler. It generalizes under compositions of the monad of exceptions with state transformer and state reader monads. For instance, in the composite monad St•Ex we have

$$?^{\mathrm{St}\bullet\mathrm{Ex}}\colon \mathrm{St}\bullet\mathrm{Ex}\,\mathrm{X}\times\mathrm{St}\bullet\mathrm{Ex}\,\mathrm{X}\to\mathrm{St}\bullet\mathrm{Ex}\,\mathrm{X}$$
$$?^{\mathrm{St}\bullet\mathrm{Ex}} = \tau^{\mathrm{St}}\circ \mathrm{map}^{\mathrm{St}}\,?^{\mathrm{Ex}}$$

When there is no ambiguity about the monad in which the exception handler is defined, we shall omit the superscript from the operator symbol '$?$'.

We are interested in the case in which the arguments each have the type T(Text). The handler satisfies the equation:

$$\begin{aligned} f\ ?\ g = \lambda xs.\ \lambda y.\ \lambda\rho.\,&\mathrm{case}\ f\ xs\ y\ \rho\ \mathrm{is}\\ &\mathit{Just}\ \xi \Rightarrow \mathit{Just}\ \xi\\ &\mid\ \mathit{Nothing} \Rightarrow g\ xs\ y\ \rho \end{aligned}$$

in which the type ExX has been specialized to the datatype introduced by Mike Spivey [5],

$$\mathit{Maybe}(\mathrm{X}) = \mathit{Just}\,(\mathrm{X}) \mid \mathit{Nothing}$$

Here, the set E of exception names is the singleton, $\{\mathit{Nothing}\}$. (The name *Nothing* replaces what we formerly called FAIL.)

In selecting one of its two arguments, the composition '$?$' favors one whose application produces an unexceptional result. Selection is sequential, trying the arguments in order from left to right. However, we may wish to describe a more complex composition not so strongly biased toward an unexceptional result. The choice among alternatives may depend upon the current attribute values through a policy predicate, $k : \mathrm{Attr}\to\mathrm{Bool}$. To express this, we introduce the combinator $?_k$ which has the same type as '$?$' but which takes the policy predicate as a (free variable) parameter. $?_k$ is no longer polymorphic with respect to the variable ρ.

$$\begin{aligned} f\ ?_k\ g = \lambda xs.\ \lambda y.\ \lambda\rho.\,&\mathrm{case}\ f\ xs\ y\ \rho\ \mathrm{is}\\ &\mathit{Just}\ \xi \Rightarrow \mathit{Just}\ \xi\\ &\mid\ \mathit{Nothing} \Rightarrow \mathrm{if}\ k\ \rho\ \mathrm{then}\ g\ xs\ y\ \rho\\ &\qquad\qquad \mathrm{else}\ \mathit{Nothing} \end{aligned}$$

Obviously, '$?$' is equivalent to '$?_{k_{tt}}$', where $k_{tt} = \lambda\rho.\ tt$.

4.5.1. Separators

A separator is a literal string that may conclude in a line break and indentation. We choose to represent a separator as a value of type

$$\mathrm{SEP} = \mathrm{String}\times\mathrm{Number}$$

where the String component represents the separator string up to the line break (possibly empty) and the Number is the length of the whitespace following the line break, as specified in the pattern. If the Number component is negative, there is no line break in the separator. We shall express this by a parameter of the composition operator, (;). Following the functional template defining the strength, τ, we define a parameterized, symmetric tensorial strength for

the composite monad constructor T:

$$\tau_h' : \mathrm{T}X \times \mathrm{T}Y \to \mathrm{T}(X \times Y)$$
$$\tau_h' = \lambda(f,g).\, \lambda xs.\, \lambda y.\, \lambda \rho.$$
$$\quad \text{case } f\ xs\ y\ \rho \text{ is}$$
$$\quad\quad Just(\ t_1, xs', y') \Rightarrow$$
$$\quad\quad\quad \text{let } y'', \rho' = h\ y'\ \rho$$
$$\quad\quad\quad \text{in case } g\ xs'\ y''\ \rho' \text{ is}$$
$$\quad\quad\quad\quad Just(t_2, xs'', y''') \Rightarrow Just((t_1, t_2), xs'', y''')$$
$$\quad\quad\quad\quad \mid\ Nothing \Rightarrow Nothing$$
$$\quad\quad \mid\ Nothing \Rightarrow Nothing$$

which is no longer polymorphic in the state variables y and ρ Note that the **let** expressions manifest the natural extensions of functions in a composite monad of state transformers and state readers. For instance, the intermediate **let** expression

$$\text{let } y'', \rho' = h\ y'\ \rho \text{ in } g\ xs'\ y''\ \rho'$$

is equivalent to:

$$((g\ xs')^* \circ h)\ y'\ \rho$$

where the natural extension is taken in the monad $\mathrm{St}\text{•}\mathrm{Rd}\,X = \mathrm{Loc} \to \mathrm{Attr} \to X \times \mathrm{Loc}$.

In terms of the parameterized strength, τ_h', we define a parameterized composition combinator

$$(\,;_h) = \lambda(f,g).\, \mathrm{T}(@) \circ \tau_h'(f,g)$$

which uses $h : \mathrm{Loc} \to \mathrm{Attr} \to \mathrm{Loc} \times \mathrm{Attr}$ as its policy function. Obviously, $(\,;) = (\,;_{\eta^{St}})$. We require that a policy function, h, must satisfy the following admissibility constraint:

(3) $$\text{if } y', \rho' = h\ y\ \rho \text{ then } y' \geq \rho'_1 \geq \rho_1$$

This constraint ensures that an indentation specification (the value of ρ_1) is always honored and that indentations are accumulated.

With the functions $literal^{\mathrm{E}}$ and $display^{\mathrm{E}}$, we can now create a composite combinator for the display of separated text terms. Define

(4)
$$separate : SEP \to \mathrm{T}(\mathrm{Text}) \times \mathrm{T}(\mathrm{Text}) \to \mathrm{T}(\mathrm{Text})$$
$$separate\ (str, indent)(f,g) = \text{let } h = \lambda y.\, \lambda \rho.\, \text{if } indent < 0 \text{ then } y, \rho$$
$$\text{else let } p = \rho_1 + indent$$
$$\text{in } p, (p, \rho_2, \rho_3)$$
$$\text{in } f\ ;\ literal^{\mathrm{E}}(compress\ str)\ ;_h\ g$$
$$\text{where } compress\ str = \text{if } is\text{-}whitespace\ str$$
$$\text{then if } newline \in str \text{ then } newline$$
$$\text{else } one\text{-}blank$$
$$\text{else } str$$

Notation: ρ_i is shorthand for $\pi_i\ \rho$.

By defining multiple policy functions, the combinator *separate* can be enriched to account for optional line breaks that may be inserted in order to satisfy a line width constraint on the output text. In the function *separate'*, the strategy is first to try the pair of layout functions with layout restricted to a single line. Thus they are tried at a stiffness level of zero. If either layout fails at this level, the next option is to retry the two layouts at the stiffness level given by the current attribute parameter, but with a line break appended to the separator string.

(5)
$$\begin{aligned} &separate'(str, indent)(f, g) = \\ &\quad \text{let } h_1 = \lambda y.\ \lambda\rho.\ y, (\rho_1, 0, \rho_3) \\ &\quad \text{and } h_2 = \lambda y.\ \lambda\rho.\ indent + \rho_1, (indent + \rho_1, \rho_2, \rho_3) \\ &\quad \text{and } k = \lambda\rho.\ \rho_2 > 0 \\ &\quad \text{in } literal^{\mathrm{E}}\,\epsilon\,;_{h_1} f\,;(literal^{\mathrm{E}}(compress\ str)\,;_{h_1} g \\ &\qquad ?_k\,(f\,;literal^{\mathrm{E}}(compress\,(str\ @\ newline))\,;_{h_2} g) \end{aligned}$$

in which the policy predicate $\rho_2 > 0$ tests that the stiffness parameter is non-zero, for otherwise, the exception should be propagated so that a line break will be made at a point less deeply nested in the term structure.

4.5.2. The display function

The function *display-item* analyzes a single abstract syntax term, discriminating on the principal constructor to display the argument list. Without the specification of optional brackets around a term variable in a pattern, the function would be simply

$$\begin{aligned} display\text{-}item &= \mathrm{red}^{AST}\,(display\text{-}args_1, \ldots, display\text{-}args_N) \\ &\qquad \circ\ map^{AST}\,((\lambda s.\ literal^{\mathrm{E}}\,s\ [\,])\circ ident\text{-}to\text{-}str) \\ &: \mathrm{AST} \rightarrow \mathrm{Loc} \rightarrow \mathrm{Attr} \rightarrow (\mathrm{Text} \times \mathrm{Loc}) + \mathrm{E} \end{aligned}$$

where red^{AST} is the reduction function for the sum-of-products datatype constructor, AST, N is the number of distinct constructors in the abstract syntax, and $display\text{-}args_i$ is the particular display function compiled from the sequence of patterns given for the i^{th} constructor.

When a display header specifies optional brackets, b_1 and b_2 around a term variable V, a customized version of *display-item* must be used. If the actual term represented by V belongs to the current conflicts set, then it must enclose the text image of this term in the required brackets. To express this, we define the customized display function by the same recursion scheme satisfied by the unfolding of red^{AST}, but which incorporates policy functions q and r.

(6)
$$\begin{aligned} &display\text{-}item_{q,r}\,(op(args))\,y\,\rho = \\ &\qquad \text{case } op \text{ is} \\ &\qquad\quad \ldots \\ &\qquad\quad \$\mathrm{ID}_i \Rightarrow \text{if } q\ \$\mathrm{ID}_i\ \rho \text{ then } r\ display\text{-}args_i\ args\ y\ \rho \\ &\qquad\qquad\qquad \text{else } display\text{-}args_i\ args\ y\ \rho \\ &\qquad\quad \ldots \end{aligned}$$

To make this function function parametric on a variable letter V and a pair of bracket strings, (b_1, b_2) define:

(7) $$q_V\ op\ \rho = op \in fst(lookup\ V\ \rho_3)$$

$$r_{b_1,b_2} f\ xs\ y\ \rho = \textbf{let}\ (b_1, b_2) = snd(lookup\ V\ \rho_3)\ \textbf{in}\ (literal^{\mathrm{E}}\ b_1\ ;\ f\ ;\ literal^{\mathrm{E}}\ b_2)\ xs\ y\ \rho$$

where $lookup\ v = snd \circ hd \circ filter^{List}\ (\lambda(x,y).\ x = v)$

The abstract syntax will have a unit constructor, which will be the unit of the monad of the AST datatype constructor. We assume the typing to be

$$\eta^{AST} : \text{Identifier} \rightarrow \text{AST}$$

For the unit constructor of the datatype AST we have that

(8) $$display\text{-}args_\eta = (\lambda s.\ literal^{\mathrm{E}}\ s\ [\,])\circ ident\text{-}to\text{-}string$$

The display of a list of terms is accomplished by $display^{\mathrm{E}}$, whose definition is:

$$display^{\mathrm{E}}\ [\,]\ y\ \rho = Just([\,], [\,], y)$$
$$display^{\mathrm{E}}(x::xs) = (\tau_1^{\mathrm{T}} \circ \eta^{\mathrm{St}}_{List(\mathrm{AST})} \circ display\text{-}item)\ x\ xs$$

where $\tau_1^{\mathrm{U}} : \mathrm{U}\,\mathrm{X}\times\mathrm{Y} \rightarrow \mathrm{U}(\mathrm{X}\times\mathrm{Y})$ is the asymmetric tensorial strength for U.

When $\mathrm{U}\,\mathrm{X} = \mathrm{Loc} \rightarrow \mathrm{Attr} \rightarrow (\mathrm{X}\times\mathrm{Loc}) + \mathrm{E}$

then $\tau_1^{\mathrm{U}} = \lambda(t,x) : \mathrm{TX}\times\mathrm{X}.\ \lambda y : \mathrm{Loc}.\ \lambda\rho : \mathrm{Attr}.\ \mathrm{case}\ t\ y\ \rho\ \mathrm{is}$

$$Just(v,y') \Rightarrow Just(v,x,y')$$
$$\mid\ Nothing \Rightarrow Nothing$$

Thus the composition $\tau_1^{\mathrm{U}} \circ \eta_{\mathrm{S}}^{\mathrm{St}}$ is the curried version of the strength,

$$\tau_1^{\mathrm{U}} \circ \eta_{\mathrm{S}}^{\mathrm{St}} = \lambda t : \mathrm{U}\,\mathrm{X}.\ \lambda x : \mathrm{X}.\ \lambda y : \mathrm{Loc}.\ \lambda\rho : \mathrm{Attr}.\ \mathrm{case}\ t\ y\ \rho\ \mathrm{is}$$
$$Just(v,y') \Rightarrow Just(v,x,y')$$
$$\mid\ Nothing \Rightarrow Nothing$$

5. Semantic translation of the layout language

We now shall give a denotational style semantics to the layout language. We assume as given a binary relation, $R \subseteq \text{OP-ID}\times\text{OP-ID}$, which corresponds to the operator precedence conflicts relation for the concrete syntax in which we are printing. If this relation is empty, the conflicts relation can be specified incrementally.

We also assume that patterns have been pre-processed to analyze each separator string into a prefix string that extends up to (and includes) any newline character, followed by any whitespace that follows a newline. The following whitespace is assumed to be represented by a numeric index, the relative indentation. Indentation is specified as tabular, with respect to characters in preceding or following lines of the same pattern. We do no specify how the indentation is to be computed. In case printing is to be done in a fixed-width font, the indentation would be simply the number of character spaces from the left margin of a pattern up to the first printing character in a line.

To translate the head specification and form a conflicts set,
define $\mathbf{H}$: HEAD $\rightarrow List(\text{VAR}\times(\text{OP-ID-LIST}\times(\text{String}\times\text{String})))$ in two steps. Let

$$\mathbf{H}\,[\![\,\text{OP–ID (ARG-LIST) CONFLICT*}\,]\!]\,R = \\ //@(\text{map}^{List}\ \mathbf{C}\,[\![\,\text{CONFLICT}\,]\!]\ (proj_{\text{OP-ID}}\,R)\,[\![\,\text{CONFLICT*}\,]\!])$$

in which CONFLICT → VAR* OP-ID-LIST BKTS, and

$$\mathbf{C}\,[\![\,\text{VAR* OP\_ID-LIST BKTS}\,]\!]\ R = \text{map}^{List}\ (\lambda v.\ v, (\text{OP-ID-LIST} \cup R, \text{BKTS}))$$

The semantics of an individual operator of the abstract syntax defines a function $display\text{-}args_i$, where i is the index of HEAD.OP-ID. For its definition we require a combinator the same in type as $\text{red}^{List}\,(\lambda xs.\,\lambda y.\,\lambda\rho.\,Nothing, (?))$. We cannot use that function, however, as the combinator for alternative layouts, '?', does not incorporate a policy function, and we require a policy that depends on the stiffness parameter of each optional pattern in turn. We define a variant of this list reduction:

$$red?\ [\,] = \lambda xs.\ \lambda y.\ \lambda\rho.\ Nothing \qquad \text{(this case should never be encountered)}$$
$$red?\ ((f, n)::others) = \text{let } k_n = \lambda\rho.\ \rho_2 > n \\ \text{in } f\ ?_{k_n}\ (red?\ others)$$

We also require a function *stiff* : String → Integer that extracts the numeral representing the stiffness parameter from an OPTION and converts it to an integer.

The compilation function that acts on layout specifications to produce display functions is $\mathbf{C}$: SPEC → T(Text):

$$\mathbf{C}\,[\![\,\text{HEAD OPTIONS}\,]\!]\ R = \text{let } Cset\text{-}list = \mathbf{H}\,[\![\,\text{HEAD}\,]\!]\ R \\ \text{and } stiffness\text{-}list = \text{map}^{List}\ stiff\ \text{OPTIONS} \\ \text{in } red?\ (zip\ (\mathbf{D}\,[\![\,\text{OPTIONS}\,]\!]\ Cset\text{-}list, stiffness\text{-}list))$$

For the i^{th} operator of the abstract syntax, the function $display\text{-}args_i$ is gotten from analysis of its patterns:

$$display\text{-}args_i = \lambda xs.\ \lambda y.\ \lambda\rho. \\ \text{case } \mathbf{C}\,[\![\,\text{HEAD}_i\ \text{OPTIONS}_i\,]\!]\ R\ xs\ y\ \rho \text{ is} \\ Just(txt, xs', y') \Rightarrow Just(txt, y') \\ \mid\ Nothing \Rightarrow Nothing$$

A list of pattern options is denoted as a list of display functions. The semantic function is

$$\mathbf{D} : \text{OPTIONS} \to List(\text{VAR} \times List(\text{OP-ID}) \times \text{String} \times \text{String}) \to \\ List(\text{AST} \to \text{Loc} \to \text{Attr} \to (\text{Text} \times List(\text{AST}) \times \text{Loc}) + \text{E})$$

$$\mathbf{D}\,[\![\,==>(n)\ \text{PATTERN OPTIONS}\,]\!]\ Csets = (\mathbf{P}\,[\![\,\text{PATTERN}\,]\!]\ n\ Csets) :: \mathbf{P}\,[\![\,\text{OPTIONS}\,]\!]\ Csets$$

$$\mathbf{D}\,[\![\,==>\ \text{PATTERN}\,]\!]\ Cset = [\mathbf{P}\,[\![\,\text{PATTERN}\,]\!]\ \infty\ Cset\,]$$

In which the pattern interpretation function (defined below) has the typing

$$\mathbf{P} : \text{PATTERN-SPEC} \to \text{Number} \to List(\text{OP-ID}) \to \text{AST} \to \text{Loc} \to \text{Attr} \to (\text{Text} \times List(\text{AST}) \times \text{Loc}) + \text{E}$$

Separators are given semantics by the function

$$\mathbf{S} : \text{SEPARATOR} \to \text{T(Text)} \times \text{T(Text)} \to \text{T(Text)}$$

whose definition is

$$\mathbf{S}\,[\![\,string\,]\!] = separate\ (string, -1)$$

$$\mathbf{S}[\![<space>]\!] = separate\,(<space>.str, <space>.indent)$$

$$\mathbf{S}[\![string\ <space>]\!] = separate\,(string@<space>.str, <space>.indent)$$

$$\mathbf{S}[\![sep\ string]\!] = \mathbf{S}[\![sep]\!] \circ (id_{\mathrm{T(Text)}} \times postseparate\,(string, -1))$$

$$\mathbf{S}[\![sep\ string\ <space>]\!] = \mathbf{S}[\![sep]\!] \circ (id_{\mathrm{T(Text)}} \times postseparate\,(string@<space>.str, <space>.indent))$$

where $postseparate\,(str, indent)\,g = separate\,(str, indent)(literal^{\mathrm{E}}\,\epsilon, g)$

Notation: $(;_S;)$ is an infix operator denoting $\mathbf{S}[\![S]\!]$ and $(;_{\overline{S}};)$ is an infix operator denoting $\mathbf{S}'[\![\{S\}]\!]$, where $\mathbf{S}'$ is defined analogously to $\mathbf{S}$ except in terms of $separate'$.

In the following, V stands for a variable letter, L for a literal string, S for a separator and ϵ for the empty string. Recall that an inherited attribute parameter has three components, the current indentation (designated by i), the stiffness level of the environment (designated by s), and mapping that associates variable names with operator conflicts sets and optional bracket pairs (designated by *Cset*).

A pattern consists of pattern elements, or a sequence of pattern elements set apart by separators. The semantics of pattern elements are:

$$\mathbf{A}[\![L]\!] = literal^{\mathrm{E}}\,L \qquad (Lit)$$

$$\mathbf{A}[\![V]\!] = display^{\mathrm{E}} \qquad (Var)$$

$$\mathbf{A}[\![V\ L]\!] = display^{\mathrm{E}}; literal^{\mathrm{E}}\,L \qquad (Terminated)$$

$$\mathbf{A}[\![L_1\ V\ L_2]\!] = literal^{\mathrm{E}}\,L_1; display^{\mathrm{E}}; literal^{\mathrm{E}}\,L_2 \qquad (Bracketed)$$

The semantic equations for patterns are:

$$\mathbf{P}[\![\varnothing]\!]\ n\ Cset = \eta^{\mathrm{St \bullet Rd \bullet E}}$$

$$\mathbf{P}[\![\text{PAT-ELT}]\!]\ n\ Cset = \lambda xs{:}List(\text{AST}).\,\lambda y{:}\text{Loc}.\,\lambda(i,s,\_){:}\text{Attr}, \mathbf{A}[\![\text{PAT-ELT}]\!]\ xs\ y\ (i, min(n,s), Cset)$$

$$\mathbf{P}[\![\text{PAT-ELT}\ S\ \text{PATTERN}]\!]\ n\ Cset = \lambda xs{:}List(\text{AST}).\,\lambda y{:}\text{Loc}.\,\lambda(i,s,\_){:}\text{Attr}. (\mathbf{A}[\![\text{PAT-ELT}]\!]\ ;_S;\ \mathbf{P}[\![\text{PATTERN}]\!]\ n\ Cset)\ xs\ y\ (i, min(n,s), Cset) \qquad (Separated)$$

$$\mathbf{P}[\![\text{PAT-ELT}\ \{S\}*]\!]\ n\ Cset = \lambda xs{:}List(\text{AST}).\,\lambda y{:}\text{Loc}.\,\lambda(i,s,\_){:}\text{Attr}. \text{let } nil = \lambda y{:}\text{Loc}.\,\lambda(i,s,\_){:}\text{Attr}.\ [\,], [\,], y \text{ in } (\mathrm{red}^{List}\,(nil, ;_S;) \circ \mathrm{map}^{List}\,(\mathbf{A}[\![\text{PAT-ELT}]\!]))\ xs\ y\ (i, min(n,s), Cset) \qquad (Iterated)$$

In (*Separated*) and (*Iterated*), when the separator indicates an optional line break, replace the separator combinator, $(;_S;)$, by $(;_{\overline{S}};)$.

6. Properties of the specification

There are several properties that can be verified by means of evidence taken directly from the specification. Here we give three of the more important ones.

Proposition 1:

All tokens of the abstract syntax are displayed exactly once.

$$\forall\, x : \mathrm{AST}.\, \forall\, y : \mathrm{Loc}.\, \forall\, \rho : \mathrm{Attr}.$$
$$\quad \mathit{display\text{-}item}\ x\ y\ \rho = \mathit{Just}(\mathit{txt}, y') \text{ and } y \geq \rho_1 \Rightarrow$$
$$\quad\quad \mathrm{mapRR}^{\mathit{List}}\ \mathit{display\text{-}atom}\ (\mathit{flatten\text{-}to\text{-}list}\ x)\ \rho_1\ \rho \subseteq^{\mathit{List}}_{1\text{-}1} \mathit{txt}$$

where $\mathit{display\text{-}atom}\ x\ y\ \rho = \mathrm{case}\ \mathit{literal}^{\mathrm{E}}(\mathit{ident\text{-}to\text{-}str}\ x)\ y\ \rho$ is

$$\mathit{Just}(\mathit{txt}, \_) \Rightarrow \mathit{txt}$$
$$\mid\ \mathit{Nothing} \Rightarrow [\text{"}\mathit{ERROR}\text{"}]$$

Here $\mathit{flatten\text{-}to\text{-}list} : \mathrm{AST} \rightarrow \mathit{List}(\mathrm{AST})$ takes the singleton nodes of an abstract syntax tree into a list of (singleton) AST's and $\mathit{literal\text{-}item} : \mathrm{AST} \rightarrow \mathrm{Loc} \rightarrow \mathrm{Attr} \rightarrow \mathrm{Text} \times \mathrm{Loc}$ is the display function for atomic AST's whose form is IDENT(str). $\mathrm{mapRR}^{\mathit{List}}$ is a map function with two state readers, specialized to *List* types. A definition is given in Appendix 1.

$$\mathit{flatten\text{-}to\text{-}list} = \mathrm{red}^{\mathit{AST}}(f_1, \ldots, f_N) \circ \mathrm{map}^{\mathit{AST}}\ \eta^{\mathit{List}}$$
$$\text{where } f_\eta = \mathit{id} \text{ and } f_i : \underbrace{\mathrm{AST} \times \cdots \times \mathrm{AST}}_{m_i} \rightarrow \mathit{List}(\mathrm{AST}) = \mathrm{red}^{m_i\text{-}\mathit{tuple}}(@)$$

(In fact, the representation chosen for an m_i-*tuple* of AST's is a list. In both the operational semantics and in proofs of properties, it is most convenient to access the elements of such a tuple sequentially, from left to right.)

Proof:

Let $\mathit{flatten\text{-}to\text{-}text} = (\mathrm{mapRR}^{\mathit{List}}\ \mathit{display\text{-}atom}) \circ \mathit{flatten\text{-}to\text{-}list}$

A parametricity theorem for lists will give a proof that

$$\mathit{flatten\text{-}to\text{-}text}\ x\ y\ \rho = \mathrm{red}^{\mathit{AST}}(f_1, \ldots, f_N)(\mathrm{mapRR}^{\mathit{AST}}\ \mathit{display\text{-}atom}\ x\ y\ \rho)$$

We need to show that

$$\forall\, x : \mathrm{AST}.\, \forall\, y : \mathrm{Loc}.\, \forall\, \rho : \mathrm{Attr}.$$
$$\quad \mathit{display\text{-}item}\ x\ y\ \rho = \mathit{Just}(\mathit{txt}, y') \Rightarrow \mathit{flatten\text{-}to\text{-}text}\ x\ y\ \rho \subseteq^{\mathit{List}}_{1\text{-}1} \mathit{txt}$$

This will be proved by induction on the structure of the type AST, since *flatten-to-text* is an instance of $\mathrm{red}^{\mathit{AST}}$.

The basis case is that the tree is IDENT(str) and

$$\mathit{display\text{-}item}\ x\ y\ \rho = \mathit{literal}^{\mathrm{E}}\ [x]\ y\ \rho$$

which either evaluates to $\mathit{Just}([\phi_{\mathrm{f}}(\mathit{str})], [], y')$ or to *Nothing*. When it yields $\mathit{Just}([\phi_{\mathrm{f}}(\mathit{str})], [], y')$, then $\mathit{display\text{-}atom}\ x\ y\ \rho = [\phi_{\mathrm{f}}(\mathit{str})]$, which validates the conclusion for the basis case.

Proof of the induction step is necessarily relative to the patterns given. The patterns must embed the abstract syntax, as a condition for well-formedness. The i^{th} case in a structural induction relies upon properties of the function, $\mathit{display\text{-}args}_i$, compiled from the patterns for display of terms whose top level operator is ID_i. Let $(\,.\,)$ be a postfix operator defined as:

$$(\,.\,) = \lambda f.\ \lambda xs.\ \lambda y.\ \lambda \rho.\ \text{case } f\ xs\ y\ \rho \text{ is}$$
$$\qquad\qquad Just(txt, xs', y') \Rightarrow Just(txt, y')$$
$$\qquad \mid\ Nothing \Rightarrow Nothing$$

We can use the representation $display\text{-}args_i = (d_1\,;_1 \cdots ;_{n_i-1} d_{n_i}\,.\,)$ in which each of the display functions d_i is either $literal^{\mathrm{E}}(str)$ or $display^{\mathrm{E}}$. Each of the connective combinators $(\,;_i\,)$ is one of $(\,;\,)$, $(\,;_h\,)$ $(\,;_s;\,)$ or $(\,;_{\bar{s}};\,)$, where h is an admissible policy function. The two latter forms of combinators are composites of $(\,;\,)$, $(literal^{\mathrm{E}}\,s)$ and $(\,;_h)$, so that we need only consider the first two forms. The first form can be considered to be a special case of the second, with $h = \eta^{\mathrm{St}}$, so we need only consider $(\,;_h)$ as the general form of the connective combinator. Remember that the admissible policy functions are constrained by the relation $y', \rho' = h\ y\ \rho \Rightarrow y' \geq \rho_1$ and $\rho'_1 \geq \rho_1$.

The composition $(d_1\,;_1 \cdots ;_{n_i-1} d_{n_i}\,.\,)$ can be analyzed as a sequence. We need the following lemma. **Notation**: let $@/\!/ = \mathrm{red}^{List}\,([\,],(@))$.

Lemma 1.1:

$$y \geq \rho_1 \Rightarrow$$
$$(d_1\,;_{h_1} \cdots ;_{h_{n_i-1}} d_{n_i}\,.\,)\,[x_1, \ldots, x_{m_i}]\ y\ \rho = Just(txt, y') \Rightarrow$$
$$\quad \mathrm{mapRR}^{List}\ display\text{-}atom\ (@/\!/(\mathrm{map}^{List}\ flatten\text{-}to\text{-}list\ [x_1, \ldots, x_{m_i}]))\,\rho_1\,\rho \subseteq {}^{List}_{1\text{-}1}txt$$

We prove the lemma by induction on n_i. $n_i \geq m_i$ so that if $n_i = 0$, then $txt = [\,]$ and the result is trivial.

Suppose $n_i = n + 1$ for some $n \geq 0$. If $d_1 = literal^{\mathrm{E}}\,str_1$ then the initial element of txt is not an image of x_1 and the conclusion of the lemma would be established if

$$(d_2\,;_{h_2} \cdots ;_{h_{n_i-1}} d_{n_i}\,.\,)\,[x_1, \ldots, x_{m_i}]\ y_1\,\rho' = Just(txt', y') \Rightarrow$$
$$\quad \mathrm{mapRR}^{List}\ display\text{-}atom\ (@/\!/(\mathrm{map}^{List}\ flatten\text{-}to\text{-}list\ [x_1, \ldots, x_{m_i}]))\,\rho_1\,\rho \subseteq {}^{List}_{1\text{-}1}txt'$$

where $y_1, \rho' = \text{let } Just(\_\,, [x_1, \ldots, x_{m_i}], y') = display^{\mathrm{E}}\,str_1\,[x_1, \ldots, x_{m_i}]\ y\ \rho$
$\qquad\qquad\qquad\qquad \text{in } h_1\,y'\,\rho$

The above implication holds by the induction hypothesis of the lemma, with $y_1 \geq \rho'_1$ and $\rho'_1 \geq \rho_1$ as consequences of the monotonicity of $literal^{E}\,s$ in its Loc-typed argument, and the policy function constraint.

If $d_1 = display^{\mathrm{E}}$ then $display\text{-}item\ x\ y\ \rho = Just(txt, y')$ if and only if both

(A) $$display\text{-}item\ x_1\,y\,\rho = Just(txt_1, y')$$

and $y_1, \rho' = h\ y'\,\rho$ in

(B) $$(d_2\,;_{h_2} \cdots ;_{h_{n_i-1}} d_{n_i}\,.\,)\,[x_2, \ldots, x_{m_i}]\ y_1\,\rho' = Just(txt', y') \Rightarrow$$
$$\quad \mathrm{mapRR}^{List}\ display\text{-}atom\ (@/\!/(\mathrm{map}^{List}\ flatten\text{-}to\text{-}list\ [x_2, \ldots, x_{m_i}]))\,\rho_1\,\rho \subseteq {}^{List}_{1\text{-}1}txt'$$

with $txt = txt_1 @ txt'$. Condition (A) holds by invoking the induction hypothesis of the proposition and (B) holds by the induction hypothesis of the lemma, with the observation that $y_1 \geq \rho'_1$ is assured by the admissibility constraint for h. The assertion that $txt = txt_1 @ txt'$ follows from the definition of the composition combinator $(\,;_h)$. This completes the proof of the lemma, from which the conclusion of the proposition follows immediately.

This completes the induction step and hence the proof of Proposition 1.
□

It would be interesting if we could prove that the prettyprinted image of an abstract syntax tree could be parsed, and that the composition of parsing with prettyprinting is the identity on abstract syntax trees. In order to do this, one would need to show that in addition to embedding the abstract syntax, the concrete syntax has an unambiguous context-free grammar. Unfortunately, there is no fully general method to do this.

The display respects layout constraints. To prove this property we shall use a modification of the predicate *isOK* that is an implicit parameter of $display^{\mathrm{E}}$. Let

(9) $$isOK'\, l\, y\, \rho = l + y \leq scroll\text{-}width$$

This test ignores maximum stiffness and fails at every detected occurrence of text overflow. Then we can state the following

Proposition 2:

$$\forall\ x : \mathrm{AST}.\ \forall\ y{:}\mathrm{Loc}.\ \forall\ \rho{:}\mathrm{Attr}.$$
$$display\text{-}item_{\mathbf{f},u,\phi,isOK'}\, x'\, y\, \rho = Just(txt, y')$$
$$\Leftrightarrow\ \forall\ txt' \subseteq^{List}_{initial} txt.\ Max\,(Lengths\,(@/\!/\ txt')\,y) \leq scroll\text{-}width$$

where $Lengths\ str\ y = fst\,(\mathrm{mapS}^{List}\ addlength\ (\mathrm{der}^{List}\ (isempty, (splitat\ newline))\ str)\ y)$

and $addlength\ str\ y = y + length_{\mathbf{f}}\, str, 0 = u_{\mathbf{f}}(y, str), 0.$

mapS^{List} and der^{List} are defined in Appendix 1.

Comment: The function $splitat : \mathrm{Char} \rightarrow \mathrm{String} \rightarrow \mathrm{String} \times \mathrm{String}$ divides its String-typed argument into the prefix and suffix, respectively, of the first occurrence of the character given as its first argument. If the string contains no occurrence of the specified character, then it pairs the original string argument with the empty string.

Thus $\mathrm{der}^{List}\ (isempty, (splitat\ newline))$ divides a string into a list of lines, each of which is a string that does not contain an occurrence of the newline character.

The function $Max = \mathrm{red}^{List}\ (0, max)$.
End of comment.

Proof: By induction on the structure of the AST.

Base step: let $x = \mathrm{IDENT}(str)$, where *str* contains no occurrence of the character *newline*. Then

(10)
$$\begin{aligned} display\text{-}item_{\mathbf{f},u,\phi,isOK'}\, x\ y\ \rho &= literal^{\mathrm{E}}_{\mathbf{f},u,\phi,isOK'}\ str\ y\ \rho \\ &= \text{if } u_{\mathbf{f}}(y, str) \leq scroll\text{-}width \\ &\qquad \text{then } Just([\phi_{\mathbf{f}}(str)], u_{\mathbf{f}}(y, str)) \\ &\qquad \text{else } Nothing \end{aligned}$$

Furthermore, $\mathrm{der}^{List}\ (isempty, (splitat\ newline))\ str = [str]$, and

$$Max(\mathrm{mapS}^{List}\ Lengths\ [\phi_{\mathbf{f}}(str)]\ y) = u_{\mathbf{f}}(y, str)$$

which relies upon the additive property $(*)$ of u. But $u_{\mathbf{f}}(y, str) \leq scroll\text{-}width$ is exactly the condition that

$$display\text{-}item_{\mathbf{f},u,\phi,isOK'}\, x\ y\ \rho = Just([\phi_{\mathbf{f}}(str)], u_{\mathbf{f}}(y, str)).$$

Induction step: let $x = \$\mathrm{ID}_i(x_1, \ldots, x_{m_i})$ and assume the hypothesis for each of $x_1, \ldots, x_{m_i}$. Then $display\text{-}item_{\mathbf{f},u,\phi,isOK'}\, x\ y\ \rho = display\text{-}args_i\ [x_1, \ldots, x_{m_i}]\ y\ \rho$. Suppose this evaluates to $Just(txt, y')$. Then no display of a component term $x_1, \ldots, x_{m_i}$ can evaluate to *Nothing*,

as the connective combinators ';' or ';$_h$' would have preserved *Nothing* as the ultimate result. As in the proof of preceding proposition, we formulate a lemma.

Lemma 2.1:

(11) $y \geq \rho_1 \Rightarrow$

$$(d_1 ;_{h_1} \cdots ;_{h_{n_i-1}} d_{n_i} \cdot)_{\mathbf{f},u,\phi,isOK'} [x_1, \ldots, x_{m_i}]\ y\ \rho = Just(txt, y')$$

$$\Leftrightarrow \forall\ txt' \subseteq^{List}_{initial} txt.\ Max\,(Lengths\,(@/\!/\ txt')\ y) \leq scroll\text{-}width$$

As in the proof of Proposition 1, the lemma is proved by induction on n_i. The basis case, for $n_i = 0$, is trivial. Suppose $n_i = n+1$. If $d_1 = literal^{\mathrm{E}}_{\mathbf{f},u,\phi,isOK'}\ str_1$ then

$$(d_1 ;_{h_1} \cdots ;_{h_{n_i-1}} d_{n_i} \cdot)_{\mathbf{f},u,\phi,isOK'} [x_1, \ldots, x_{m_i}]\ y\ \rho = Just(txt, y')$$

if and only if both

$$literal^{\mathrm{E}}_{\mathbf{f},u,\phi,isOK'}\ str_1 = Just(txt_1, y_1)$$

in which $txt_1 = [\phi_{\mathbf{f}}(str_1)]$, and letting $y'_1, \rho' = h_1\, y_1\, \rho$,

$$(d_2 ;_{h_2} \cdots ;_{h_{n_i-1}} d_{n_i} \cdot)_{\mathbf{f},u,\phi,isOK'} [x_1, \ldots, x_{m_i}]\ y'_1\ \rho' = Just(txt', y')$$

where $txt = txt_1 @ txt'$. There are two cases to consider. If str_1 contains no occurrence of the *newline* character, then txt_1 does not end a line, and so the additive property of $u_{\mathbf{f}}$ assures that

$$Max\,(Lengths\,(@/\!/\ txt)) = Max\,(Lengths\,(@/\!/\ txt')).$$

By the list-induction hypothesis of the lemma, the conclusion holds.

If on the other hand, str_1 does contain an occurrence of *newline*, it must be the final character, and $Lengths\ txt_1 = length_{\mathbf{f}}(str_1/newline)$, where the notation str/c denotes the string str less its terminal substring c. Also,

$$Max\,(Lengths\,(@/\!/\ txt)) = \max(length_{\mathbf{f}}(str_1/newline), Lengths\ (@/\!/\ txt')).$$

But $y\mathbf{f}(y, str_1/newline) \leq scroll\text{-}width$ is exactly the condition that

(12) $$literal^{\mathrm{E}}_{\mathbf{f},u,\phi,isOK'}\ str_1\ xs\ y\ \rho = Just([\phi_{\mathbf{f}}(str_1)], xs, \rho_1).$$

In conjunction with the list induction hypothesis, this entails the conclusion of the lemma.

The case remaining to be considered is that in which $d_1 = display^{\mathrm{E}}_{\mathbf{f},u,\phi,isOK'}$. The argument in this case is similar to that given above, but makes use of the induction hypothesis of the main proposition as well as that of the lemma, and is omitted.

Thus we conclude the lemma by list induction, and from the lemma, the proposition follows by induction on the structure of abstract syntax trees.

□

Proposition 3: *The display respects indentation specifications.*

(a) Indentation is unchanged by display of a term or a literal string.

(b) A PATTERN following a PATTERN-ELEMENT is vertically aligned.

(c) Indentation accumulates when PATTERN-ELEMENTs are nested or iterated.

(d) Indentation whitespace occurs only after a newline character.

Proof:

(a) is a consequence of the fact that the indentation parameter is a component of read-only state for the functions $literal^{E}$ and $display^{E}$.

(b) follows directly from the definition of $\mathbf{D}\,[\![\text{PAT-ELT PAT}]\!]$ and of the separator connective $(;_s;)$.

(c) Nested and iterated PATTERN-ELEMENTs are defined by semantic equations (*Separated*) and (*Iterated*), which use either the separator combinator $(;_s;)$ or $(;_s^{=};)$. Equation (3) assures that the policy function h in the definition of $separate'$ increments the indentation parameter by the indentation specified in the separator, and (i) $(;_h)$ sets the current horizontal position equal to the new indentation; (ii) $(;_h)$ propagates the new indentation to the layout of a following PATTERN-ELEMENT.

(d) The horizontal position is reset to the value of the indentation parameter only by the separator combinator. We rely upon the translation of the layout language to correctly set the indentation value corresponding to each separator string found in a pattern.
□

7. Extending the Specification

As a non-trivial illustration of design maintenance, we consider an extension of the original specification that allows indentations to be computed, rather than directly interpreted as the number of initial blank spaces on a line in a layout pattern. The modification involves changing the type of a state component, and requires change to every policy function whose type depends upon the modified type. We shall reap a big benefit by having identified all policy parameters of the combinators, even though that may seem to have added complex notation in the initial design.

In the specification as given to this point, the indentation to be inserted following a line break is computed directly from a pattern. It is simply the number initial blank characters on a pattern line, following a line break. This is adequate when printing in a fixed character-width font such as the `tt`-font in which the patterns are specified, and with the added constraint that no text variable occurs in the prefix of the line that lies directly above the indentation. If a variable does occur in the prefix over which the indentation is measured, as in the pattern

```
if x then y
     else z
```

then the printed image may not look as intended, even if the font is fixed-width, as in

```
if null xs then u
     else rev (x::u) xs
```

In case the font is not fixed-width, the appearance will be even less predictable.

if null xs **then** u
  else rev (x::u) xs

To overcome this difficulty, we propose a further extension of the specification. The idea is simple. Character positions in a pattern should be treated as tabs. An indentation that is specified in a pattern as, for instance, three blank characters will indicate an advance by three tab stops. The actual tab settings to be used cannot be calculated from a pattern, but must be calculated dynamically. In order to accommodate this calculation, it will no longer be sufficient to let Loc, the position attribute type, be a single number; it must be a list of numbers corresponding to the tab positions calculated in a line. We redefine

$$\text{Loc} = List(\text{Num})$$

and the first component of inherited attributes now assumes the type

$$\rho_1 : \text{Int}$$

and represents an index for a value of type Loc, rather than the value of an indentation offset. Each time a new item is printed in a line, this state component of type Loc will be modified by extending the list of defined tab positions. The most recent tab position added to the list will be the offset of the current location.

This extension to the definition is a major one. Nevertheless, it is easy to see which functions must be modified. They will be the policy functions whose types include Loc. The definition of $literal^{E}_{isOK,\mathbf{f},\phi,u}$ uses the policy function u to update the current location, $u : \text{String} \times \text{Loc} \rightarrow Loc$. Since a literal string occurs fully elaborated in a pattern, a tab setting must be calculated for each character. We replace the old policy function with a new one, which calculates a list of tab settings, analyzing an argument string character by character:

(1′)
$$\begin{aligned} & u_{\mathbf{f}}(\epsilon, ys) = ys \\ & \mid u_{\mathbf{f}}(c.cs, [\,]) = u_{\mathbf{f}}(cs, [length_{\mathbf{f}}(c)]) \\ & \mid u_{\mathbf{f}}(c.cs, (y::ys)) = u_{\mathbf{f}}(cs, (y+length_{\mathbf{f}}(c)::y::ys)) \end{aligned}$$

The policy predicate *isOK* must also be redefined to take account of the revised type Loc by extracting a numeric value from a list with *hd*.

(2′)
$$\begin{aligned} & isOK\ s\ y\ \rho = hd(u_{\mathbf{f}}(y,s),0) \leq scroll\_width \text{ orelse } \rho_2 = \infty \\ & \text{where } hd([\,], i) = i \\ & \qquad hd(x::xs, i) = x \end{aligned}$$

The admissibility constraint on the policy function h that occurs in the definition of the sequential pattern composition operator, $;_h$, is

(3′)
$$\text{if } y', \rho' = h\ y\ \rho \text{ then } hd(y',0) \geq \rho'_1 \geq \rho_1$$

The definition of h that appears in *separate* becomes

(4′)
$$\begin{aligned} h\ y\ \rho = & \text{ if } indent < 0 \text{ then } y, \rho \\ & \text{else let } \rho'_1 = \rho_1 + indent \\ & \quad \text{in } suffix\ y\ \rho'_1, (\rho'_1, \rho_2, \rho_3) \end{aligned}$$

in which
$$\begin{aligned} suffix\ ys\ n = & \text{let } m = length\ ys \\ & \text{in if } m > n \text{ then } tls\,(m-n)\,ys \text{ else } ys \\ & \\ & \text{where } tls\ 0\ ys = ys \\ & \quad \mid tls\ m\ [\,] = [\,] \\ & \quad \mid tls\ m\ (y::ys) = tls\ (m-1)\ ys \end{aligned}$$

The definition of h_1 in *separate′* (equation 5) is unchanged from the original version, as it does not analyze or modify its argument of type Loc, but h_2 in *separate′* becomes the function h defined above.

The function *display-item* displays a term that is represented in a pattern by a single variable letter. Thus the analysis of a pattern will only have allocated a single tab stop to be set with the length added to a display by this term. However, the definition of *display-item* uses $literal^{E}$, which allocates a number of tab stops equal to the length of its string argument. To reduce the number of added tab stops to one, we need to replace all uses of *display-item* by uses of *display-item′*, whose definition is

(6′)
$$display\text{-}item'\ x\ y\ \rho = (f\ y)^{*E}(display\text{-}item\ x\ y\ \rho)$$

where

$$f : \mathrm{Loc} \rightarrow (\mathrm{Text} \times \mathrm{Loc}) \rightarrow \mathrm{E}(\mathrm{Text} \times \mathrm{Loc})$$
$$f\ y\ (t, y') = \mathit{Just}(t, hd(y', 0) :: y)$$

The structure of our specification has allowed us to undertake a significant modification with some degree of confidence. Isolation of policy decisions taken in the design, by specifying policy functions as parameters of otherwise generic combinators, has allowed us to locate those parameters that may be affected by a proposed modification. The type signatures of policy functions provides a search key to use in locating the affected ones. The only exception to this rule was the implicit dependence *display-item* on the assumed structure of the type Loc. This was not an explicit dependence because it only showed up because the original definition violated an expected invariant of the *modified* specification, an invariant that was not needed to verify the original specification. This invariant is that the length of a list of location numbers (a value of type Loc) computed for display of any item corresponds to the count of symbols in an initial sequence of a line of a pattern, including the symbol representing the item.

7.1. Re-verification of properties

Having extended the specification to accommodate more sophisticated calculation of indentation offsets, we must reestablish that key properties of the original specification still hold.

Every term of abstract syntax is displayed exactly once. The proof given for Proposition 1 does not depend upon the policy functions u or *isOK*. Its only explicit dependence on the structure of the type Loc comes in the 'guard' condition in the hypothesis of Proposition 1 and again in Lemma 1.1, namely that $y \geq \rho_1$, where y : Loc. Wherever this condition is referred to in Proposition 1 and its proof, it should be replaced by $hd(y,0) \geq \rho_1$.

The display respects layout constraints. Proposition 2 does use properties of the policy functions modified in extending the specification and thus must be reviewed. We see that the proof uses only the additive property of u, namely that $u_{\mathbf{f}}(y, s_1@s_2) = u_{\mathbf{f}}(u_{\mathbf{f}}(y, s_1), s_2)$. This property is easily proved of the redefined function u by induction on the number of characters in the string s_2.

There is also a comparison of the horizontal position parameter with the scroll width in equation (10) in the proof of Lemma 2.1. This equation is modified to:

(10′) $$\begin{aligned} \mathit{display\text{-}item}_{\mathbf{f},u,\phi,isOK'}\ x\ y\ \rho &= \mathit{literal}^{\mathrm{E}}_{\mathbf{f},u,\phi,isOK'}\ \mathit{str}\ y\ \rho \\ &= \text{if } hd(u_{\mathbf{f}}(y, \mathit{str}), 0) \leq \mathit{scroll\text{-}width} \\ &\qquad \text{then } \mathit{Just}([\phi_{\mathbf{f}}(\mathit{str})], u_{\mathbf{f}}(y, \mathit{str})) \\ &\qquad \text{else } \mathit{Nothing} \end{aligned}$$

In the proof of Proposition 2, Lemma 2.1 also refers to ρ_1, the first component of the inherited attribute. In the original design, this is of type Num and represents the current indentation offset. In the modified design, its type is Int and it represents an index into the list of tab stops, of type Loc. The condition in the first line of the statement of Lemma 2.1 becomes

(11′) $$hd(y,0) \geq hd(\mathit{suffix}\ y\ \rho_1, 0) \Rightarrow$$

and equation (12) becomes

(12′) $$\mathit{literal}^{\mathrm{E}}_{\mathbf{f},u,\phi,isOK'}\ \mathit{str}_1\ xs\ y\ \rho = \mathit{Just}([\phi_{\mathbf{f}}(\mathit{str}_1)], xs, \mathit{suffix}\ y\ \rho_1).$$

which also entails the conclusion of the lemma. Thus we have checked that the proof of Proposition 2, as modified, still holds.

The display respects indentation specifications. In the informal proof given for Proposition 3, clause (c) refers to the update of the indentation parameter and calculation of a new

indentation offset by the combinator *separate'*. Recall that the the calculation of the offset was modified to use a redefined policy function here.

8. Conclusions and Further Work

When giving a specification of a complex system, it is of crucial importance to have in mind a structure that underlies the system. If specification is at an abstract level, this is likely to be a mathematical structure that does not necessarily conform to the decomposition by function customarily sought by a software engineer. For the example reported here, we have made use of a mathematical structure that enabled us to build up the required state and control in a systematic, clear and rigorous way. This mathematical structure is a composition of monads, an idea suggested by E. Moggi [3] but not previously applied to the specification of software, insofar as the author is aware.

This specification makes extensive use of an exception mechanism to specify backtracking control. Using the structure imposed by monads has been enormously helpful to reason about a semantic structure that is, superficially at least, quite complex, utilizing mutable state (the current location), inherited but locally immutable state, and exceptions. Yet the monad structure has enabled us to define simple and regular composition operators and to reason confidently about composite computations.

Experience with this example supports our cautious optimism that design changes can be made safely within the design framework illustrated here. In our actual experience, the original design document was altered in two respects in order to make these changes feasible.

(i) The policy function u_f was introduced explicitly; in an original version of the document, this policy for updating horizontal position was left implicit, in the form of in-line expressions appearing in the definition of the combinator $literal^E$ and in the proof of Proposition 2. Implicit policy functions are incompatible with our design methodology, however. Editing the original document to make this parameter explicit corrected an inconsistency in applying the methodology.

(ii) The equations in which policy parameters were used or defined had not been numbered in the original document. They were numbered before attempting the revision, to make it easier to refer to the original design.

The specification has been developed without a prototype implementation. For this problem, early functional prototyping did not seem necessary, since it would be used principally to gain confidence in layout policy decisions that we have incorporated as parameters. However, the specification has undergone refinement during the process of developing proofs of its most important properties. This has allowed us to find sometimes subtle errors in the original specification and forced us to refine details. Verification of properties is the most effective and rigorous method available for refining and validating a specification.

The next step in this project will be to construct a prototype implementation. It will be composed of the generic (i.e. polymorphic) functions defined in the specification, that is, the maps, units, reductions and strengths of the constituent monads that structure the computational domain. These functions will be generated automatically by meta-programs that interpret the monad definitions and compositions. Other functions that are not generic, those that incorporate policy functions, may also be generated by specifying the point at which policy is to be applied, but we are as yet unsure to what degree these can be automated. Our goal is to generate nearly all of the operational semantics of the layout language interpreter without hand coding.

Acknowledgements

During the course of this research, the author has had many productive discussions with his colleagues Jim Hook and Tim Sheard, that have helped immeasurably to refine ideas about

how to specify and what to specify.

References

[1] Hughes, J., "A Functional Prettyprinter," Notes from a Workshop, Banff, 1991.

[2] MacLane, S., *Categories for the Working Mathematician*, Springer-Verlag, 1971.

[3] Moggi, E., "An abstract view of programming languages," LFCS-90-113, Department of Computer Science, University of Edinburgh, 1990.

[4] Oppen, D., "Prettyprinting," *ACM Trans. Prog. Lang. and Systems*, vol. 2, 4 (1980), pp. 465-483.

[5] Spivey, M., "A functional theory of exceptions," *Science of Computer Programming*, vol. 14, 1 (1990), pp. 25-42.

[6] Wadler, P., "Comprehending monads," *Proc. 1990 ACM Sympos. on Lisp and Functional Programming*, 1990, pp. 61-78.

Appendix 1: An inventory of functions needed to implement a prettyprinter

Generic function constructors

For each monad object mapping, T,

$\text{unit}^{\text{T}}, \eta^{\text{T}}_{\text{X}} : \text{X} \rightarrow \text{T X}$

$\text{map}^{\text{T}} : (\text{X} \rightarrow \text{Y}) \rightarrow \text{T X} \rightarrow \text{T Y}$

strength, $\tau^{\text{T}} : \text{T X} \times \text{T Y} \rightarrow \text{T}(\text{X} \times \text{Y})$ where
$\text{T X} = \text{S} \rightarrow \text{X} \times \text{S}$
$\text{T X} = \text{X} + \text{E}$

'$?$': $(\text{T} \bullet \text{E})\text{X} \times (\text{T} \bullet \text{E})\text{X} \rightarrow (\text{T} \bullet \text{E})\text{X}$—left-biased selection composition, where E is the monad of exceptions,

and for the monad of an additive datatype constructor, T,

$$\text{red}^{\text{T}}(f_1, \ldots, f_N) : \text{T X} \rightarrow \text{X}$$

where N is the number of constructors of the sum-of-products datatype T, and if the i^{th} constructor has type $\text{T}_1 \times \cdots \times \text{T}_{\text{m}_i} \rightarrow \text{T X}$, then the i^{th} element of the argument tuple has the typing $f_i : \text{T}_1 \times \cdots \times \text{T}_{\text{m}_i} \rightarrow \text{X}$.

The combinators '$?$' and ';' and the reduction functions must be modified to produce policy-driven variants, '$?_k$', '$;_h$' *red?* and $display^{\text{E}}_{(q,r)}$. These are no longer generic functions, however they follow the same recursion schemes as do the unfoldings of the generic functions '$?$', ';', red^{List} and red^{AST}.

Other generic functions that can be produced with compositions of the primitive ones include:

List-strength:

$$\tau^{\text{T}/List} = \text{red}^{List}((\eta^{\text{T}}\ [\,]), \tau^{\text{T}})$$

where τ^{T} is the symmetric tensorial strength for T.

Some useful polymorphic functions for lists

List map with state transformer:

$$\text{mapS} : (\text{X} \rightarrow \text{S} \rightarrow \text{Y} \times \text{S}) \rightarrow List(\text{X}) \rightarrow S \rightarrow List(\text{Y}) \times \text{S}$$
$$\text{mapS}\, f = \tau^{\text{St}/List} \circ \text{map}^{List}\, f$$

which satisfies the recursion equations:

$$\text{mapS}\, f\ [\,]\ s = [\,], s$$
$$\text{mapS}\, f\ (x :: xs)\, s = \text{let } y, s' = f\ x\ s \text{ in}$$
$$\text{let } ys, s'' = \text{mapS}\, f\ xs\ s' \text{ in}$$
$$(y :: ys), s''$$

List map with two state readers:

$$\text{mapRR} : (\text{X} \rightarrow \text{R} \rightarrow \text{S} \rightarrow \text{Y} \times \text{S}) \rightarrow List(\text{X}) \rightarrow R \rightarrow S \rightarrow List(\text{Y}) \times \text{R} \times \text{S}$$
$$\text{mapRR}\, f = \text{map}^{\text{Rd}_\text{R}}(\tau^{\text{Rd}_\text{S}/List}) \circ \tau^{\text{Rd}_\text{R}/List} \circ \text{map}^{List}\, f$$

which can also be expressed with lambda-abstraction as:

$$\text{mapRR}\, f\ r\ s = map^{List}\, (\lambda x.\ f\ x\ r\ s)$$

The list maps can be extended to arbitrary sum-of-products types.

The *List-anamorphism* constructor is

$$\begin{aligned}&\mathrm{der}^{List} : ((\mathrm{X} \rightarrow \mathrm{Bool}) \times (\mathrm{X} \rightarrow \mathrm{Y} \times \mathrm{X})) \rightarrow \mathrm{X} \rightarrow List(\mathrm{Y})\\&\mathrm{der}^{List}\,(p, f)\,x = \text{if } p\ x \text{ then } [\,]\\&\qquad\qquad\qquad\qquad \text{else } (cons \circ (id_{List(\mathrm{Y})} \times \mathrm{der}^{List}\,(p, f)) \circ f)\,x\end{aligned}$$

(In the LML library, there is a function called *choplist* which is related to the List-anamorphism by $choplist\ f = \mathrm{der}^{List}\,(isempty, f)$.)

To complete the inventory of meta-programming functions it is also necessary to have a *parser* for the layout specifications language and a variety of functions that interpret the abstract syntax of the layout language to generate a prettyprinter.

Appendix 2: Abstract Syntax of the Layout Language

Sorts: {SPEC,LAYOUT,HEAD,OPT,OPTS,ARG,CONF,OPS,OP-ID,PAT,PAT-ELT,SEP,LETTER,STRING}

Subsorts: OP-ID⊂OPS
LETTER⊂ARG,PAT-ELT
STRING⊂PAT-ELT,SEP
PAT-ELT⊂PAT

Signatures:

SPEC
Spec: LAYOUT*

LAYOUT
Layout: HEAD×OPT

HEAD
Head: OP-ID×ARG×CONF*

ARG
Args: ARG*
List-arg: LETTER

OPT
Option: NUMBER×PAT

OPTS
Options: OPT*

CONF
Conflict: ARG×OPS×STRING×STRING

OPS
Op-ids: OP-ID*

PAT

Separated:	PAT×SEP×PAT	accounts for line breaks and other separators
Iterated:	PAT×SEP	display of term lists

SEP

White:	STRING	may contain line break & indentation
Postwhitesep:	STRING×STRING	whitespace follows a visible string
Wide:	SEP×STRING	allows whitespace prefixed to visible string
Postwhitewide:	SEP×STRING×STRING	allows postfixed whitespace as well

PAT-ELT

Terminated:	PAT-ELT×STRING	when a literal follows a term
Bracketed:	STRING×PAT-ELT×STRING	no separator between brackets and term