A Type-and-Affect System for Semisignificant Whitespace

Stefan Muller
Illinois Institute of Technology

```
if (p != NULL) {
  printf("%d\n", *p)
}
```

```
if (p != NULL)
{
  printf("%d\n", *p)
}
```

Figure 1: If you're a C programmer, one of these looks beautiful and one of them makes you want to vomit.

```
if (p != None):
  print("p is " + p + " and oh no I'm getting close to the end of the line can I put a linebreak here? I don't know")
```

Figure 2: Python syntax terrifies me.

$$
\begin{array}{rcl}
\tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \tau \times \tau \mid \tau \to \tau \\
\underline{\nu} & ::= & \top \mid \bot \mid \mathrm{I} \mid \dagger \mid \emptyset \\
e & ::= & \overline{n} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{fun}\ x\ \_ \to e \mid e\ e \mid (e, e) \\
 & & \mid \mathsf{let}\ (x, y) = e\ \mathsf{in}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & & \mid \_e \mid e\_ \mid \backslash n e \mid e \backslash n
\end{array}
$$

Figure 3: The syntax of Pedant

ABSTRACT

Whitespace characters are generally ignored by compilers for most languages. But for something ignored by the compiler, programmers sure do spend a lot of time arguing about the use of whitespace in programs. In this paper, we show whitespace some (tough?) love and enforce some of your favorite pedantic style conventions statically. Oh, you asked if this is like a linter or something? Not exactly...

1 INTRODUCTION

Most languages (with the notable exception of Whitespace [3]) treat whitespace characters as insignificant other than as delimiters between tokens, and discard them during lexing. That's right, whitespace characters don't even make it to the parser, that's how overlooked they are. And yet, whitespace is related to so many of the features that many programmers and programming instructors spend so much time lecturing and/or arguing about: every programming course, company, book author, and pedant in general has a style guide for their preferred language. Many of these style guides give advice on line length, indentation, when to start a new line before and after delimiters, and other features directly or indirectly related to whitespace (newlines, spaces, and, if you're a terrible person, tabs), as seen in Figure 1. Some provide good advice on creating reasonable code, others harp on pet peeves, and many are contradictory (e.g., [2, 4]). Few languages make these guidelines standard or enforceable, and some languages even make good style difficult. It is an irony of programming languages that languages which do not have "significant whitespace" have no way of enforcing good style. Meanwhile, languages in which whitespace characters are signficant can enforce some stylistic conventions (e.g., indentation in Python), but the very fact that whitespace is significant makes it difficult to achieve other elements of good style (who knows when you can break a long line in a Python program without wreaking havoc?, Figure 2).

In this paper, we begin by presenting a type system Pedant that enforces good coding style with regard to whitespace, drawing on the best of the significant and insignificant whitespace worlds. In a normal paper introduction, we would now go on to describe how we do this and the results we achieve, but as a result, most academic papers are very uniform as to narrative style. It would be like if every novel took the approach of beginning the first chapter with the protagonists reflecting back on the events that are to be described in the novel. This can be a good storytelling technique, but it doesn't work well if overused. Instead, I'll start with a more traditional narrative style that builds a higher degree of suspense. And so we set off on this adventure together.

2 PEDANT: A SYNTAX AND TYPE SYSTEM FOR ENFORCING PEDANTIC STYLE RULES

In this section, we begin to develop Pedant, a language equipped with a type system to enforce the pedantic style rules we discussed in the introduction. Figure 3 gives the syntax of Pedant, an ML-style calculus extended with visible whitespace (␣, \n). The calculus consists of integers, Booleans, lambdas, pairs, and the relevant elimination forms (including let-binding for pairs).

The typing judgment for Pedant is $\Gamma \vdash e : {}_{m}\tau^{\underline{\nu}}_{n}$, meaning that under variable context Γ, the expression e has type τ, starts with m columns of whitespace, and is at most n columns wide. The type also includes a description $\underline{\nu}$ of e's newlines, which can be $\top$ ("top", starts with a newline), $\bot$ ("bottom", ends with a newline), I ("both", starts and ends with a newline), † ("internal", does not start or end with a newline but is not all on one line), and $\emptyset$ ("none", is all on one line).

Figure 4 gives the typing rules for Pedant. The variable rule T-Var looks x up in the context and types x with the given type. Its width is $|x|$, the number of characters in the variable name. We assume variable names do not contain whitespace, so the left is 0 and the newline description is $\emptyset$. The introduction rules for integers and booleans are straightforward as well. The remaining rules are standard as far as the introduction and elimination of types τ goes, but the whitespace-relevant parts of the rules are somewhat[1] nonstandard, and so we'll describe them very briefly before going way too quickly on to the next section of the paper. As an example, there are three rules for "if". Rule bool-E1 assumes that

SIGBOVIK 2022,
2022.

[1] I'm hoping there will be an award for "biggest understatement" this year

$$\frac{}{\Gamma, x : \tau \vdash x : {}_0\tau^{\emptyset}_{|x|}}\ (\textsc{Var}) \qquad \frac{}{\Gamma \vdash \overline{n} : {}_0\mathsf{int}^{\emptyset}_{|\overline{n}|}}\ (\text{int-I}) \qquad \frac{}{\Gamma \vdash \mathsf{true} : {}_0\mathsf{bool}^{\emptyset}_{4}}\ (\text{bool-I1}) \qquad \frac{}{\Gamma \vdash \mathsf{false} : {}_0\mathsf{bool}^{\emptyset}_{5}}\ (\text{bool-I2})$$

$$\frac{\Gamma, x : \tau_1 \vdash e : {}_{m+1}\tau_2{}^{\emptyset}_{n}}{\Gamma \vdash \mathsf{fun}\,x\,{-}{>}\,e : {}_0\tau_1 \to \tau_2{}^{\emptyset}_{4+|x|+4+m+n}}\ (\to\text{-I1}) \qquad \frac{\Gamma, x : \tau_1 \vdash e : {}_{m+1}\tau_2{}^{\top}_{n}}{\Gamma \vdash \mathsf{fun}\,x\,{-}{>}\,e : {}_0\tau_1 \to \tau_2{}^{\dagger}_{\max(m+1+n,4+|x|+4)}}\ (\to\text{-I2})$$

$$\frac{\Gamma \vdash e_1 : {}_m\tau_1 \to \tau_2{}^{\emptyset}_{n} \qquad \Gamma \vdash e_2 : {}_j\tau_1{}^{\emptyset}_{k}}{\Gamma \vdash e_1\,e_2 : {}_m\tau_2{}^{\emptyset}_{n+1+j+k}}\ (\to\text{-E1}) \qquad \frac{\Gamma \vdash e_1 : {}_m\tau_1 \to \tau_2{}^{\bot}_{n+1} \qquad \Gamma \vdash e_2 : {}_{m+1}\tau_1{}^{\dagger}_{n}}{\Gamma \vdash e_1\,e_2 : {}_m\tau_2{}^{\dagger}_{n+1}}\ (\to\text{-E2})$$

$$\frac{\Gamma \vdash e_1 : {}_m\tau_1 \to \tau_2 l\Phi^{\dagger}_{n+1} \qquad \Gamma \vdash e_2 : {}_{m+1}\tau_1{}^{\top}_{n}}{\Gamma \vdash e_1\,e_2 : {}_m\tau_2{}^{\dagger}_{n+1}}\ (\to\text{-E3}) \qquad \frac{\Gamma \vdash e_1 : {}_m\tau_1{}^{\emptyset}_{n} \qquad \Gamma \vdash e_2 : {}_j\tau_2{}^{\emptyset}_{k}}{\Gamma \vdash (e_1, e_2)_0\tau_1 \times \tau_2{}^{\emptyset}_{m+n+j+k+3} :}\ (\times\text{-I1}) \qquad \frac{\Gamma \vdash e_1 : {}_m\tau_1{}^{\dagger}_{n} \qquad \Gamma \vdash e_2 : {}_{m+1}\tau_2{}^{\top}_{n}}{\Gamma \vdash \mathsf{true} : {}_0\tau_1 \times \tau_2{}^{\dagger}_{m+n+1}}\ (\times\text{-I2})$$

$$\frac{\Gamma \vdash e_1 : {}_m\tau_1 \times \tau_2{}^{\emptyset}_{n} \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : {}_{j+1}\tau'{}^{\emptyset}_{k}}{\Gamma \vdash \mathsf{let}\,(x, y) = e_1\,\mathsf{in}\,e_2 : {}_0\tau'{}^{\emptyset}_{5+|x|+2+|y|+3+m+n+3+j+1+k}}\ (\times\text{-E1}) \qquad \frac{\Gamma \vdash e_1 : {}_m\tau_1 \times \tau_2{}^{\emptyset}_{n} \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : {}_{j+1}\tau'{}^{\mathrm{I}}_{k}}{\Gamma \vdash \mathsf{let}\,(x, y) = e_1\,\mathsf{in}\,e_2 : {}_0\tau'{}^{\dagger}_{\max(5+|x|+2+|y|+3+m+n+3, j+1+k)}}\ (\times\text{-E2})$$

$$\frac{\Gamma \vdash e_1 : {}_{m+1}\tau_1 \times \tau_2{}^{\mathrm{I}}_{n} \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : {}_{m+1}\tau'{}^{\mathrm{I}}_{n}}{\Gamma \vdash \mathsf{let}\,(x, y) = e_1\,\mathsf{in}\,e_2 : {}_0\tau'{}^{\dagger}_{\max(5+|x|+2+|y|+3, m+1+n)}}\ (\times\text{-E3}) \qquad \frac{\Gamma \vdash e_1 : {}_{m+1}\mathsf{bool}^{\emptyset}_{n} \qquad \Gamma \vdash e_2 : {}_{j+1}\tau^{\emptyset}_{k} \qquad \Gamma \vdash e_3 : {}_{i+1}\tau^{\emptyset}_{l}}{\Gamma \vdash \mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3 : {}_0\tau^{\emptyset}_{2+m+1+n+5+j+1+k+5+i+1+l}}\ (\text{bool-E1})$$

$$\frac{\Gamma \vdash e_1 : {}_{m+1}\mathsf{bool}^{\emptyset}_{n} \qquad \Gamma \vdash e_2 : {}_{j+1}\tau^{\mathrm{I}}_{k} \qquad \Gamma \vdash e_3 : {}_{j+1}\tau^{\top}_{k}}{\Gamma \vdash \mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3 : {}_0\tau^{\dagger}_{\max(2+m+1+n, j+1+k)}}\ (\text{bool-E2}) \qquad \frac{\Gamma \vdash e_1 : {}_{m+1}\mathsf{bool}^{\mathrm{I}}_{n} \qquad \Gamma \vdash e_2 : {}_{m+1}\tau^{\mathrm{I}}_{n} \qquad \Gamma \vdash e_3 : {}_{m+1}\tau^{\top}_{n}}{\Gamma \vdash \mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3 : {}_0\tau^{\dagger}_{\max(4, m+1+n)}}\ (\text{bool-E3})$$

$$\frac{\Gamma \vdash e_1 : {}_m\tau_1{}^{\emptyset}_{n} \qquad \Gamma, x : \tau_1 \vdash e_2 : {}_{j+1}\tau_2{}^{\emptyset}_{k}}{\Gamma \vdash \mathsf{let}\,x = e_1\,\mathsf{in}\,e_2 : {}_0\tau'{}^{\emptyset}_{4+|x|+2+m+n+3+j+1+k}}\ (\textsc{Let1}) \qquad \frac{\Gamma \vdash e_1 : {}_m\tau_1{}^{\emptyset}_{n} \qquad \Gamma, x : \tau_1 \vdash e_2 : {}_{j+1}\tau_2{}^{\mathrm{I}}_{k}}{\Gamma \vdash \mathsf{let}\,x = e_1\,\mathsf{in}\,e_2 : {}_0\tau'{}^{\dagger}_{\max(4+|x|+2+m+n+3, j+1+k)}}\ (\textsc{Let2})$$

$$\frac{\Gamma \vdash e_1 : {}_{m+1}\tau_1{}^{\mathrm{I}}_{n} \qquad \Gamma, x : \tau_1 \vdash e_2 : {}_{m+1}\tau_2{}^{\mathrm{I}}_{n}}{\Gamma \vdash \mathsf{let}\,x = e_1\,\mathsf{in}\,e_2 : {}_0\tau'{}^{\dagger}_{\max(4+|x|+2, m+1+n)}}\ (\textsc{Let3}) \qquad \frac{\Gamma \vdash e : {}_m\tau^{\emptyset}_{n}}{\Gamma \vdash \,e : {}_{m+1}\tau^{\emptyset}_{n}}\ (\textsc{Space1}) \qquad \frac{\Gamma \vdash e : {}_m\tau^{\emptyset}_{n}}{\Gamma \vdash e\, : {}_m\tau^{\emptyset}_{n+1}}\ (\textsc{Space2}) \qquad \frac{\Gamma \vdash e : {}_m\tau^{\dagger}_{n}}{\Gamma \vdash \backslash\mathsf{n}e : {}_m\tau^{\top}_{n}}\ (\textsc{Newline1})$$

$$\frac{\Gamma \vdash e : {}_m\tau^{\bot}_{n}}{\Gamma \vdash \backslash\mathsf{n}e : {}_m\tau^{\mathrm{I}}_{n}}\ (\textsc{Newline2}) \qquad \frac{\Gamma \vdash e : {}_m\tau^{\dagger}_{n}}{\Gamma \vdash e\backslash\mathsf{n} : {}_m\tau^{\bot}_{n}}\ (\textsc{Newline3}) \qquad \frac{\Gamma \vdash e : {}_m\tau^{\top}_{n}}{\Gamma \vdash e\backslash\mathsf{n} : {}_m\tau^{\mathrm{I}}_{n}}\ (\textsc{Newline2}) \qquad \frac{\Gamma \vdash e : {}_m\tau^{\nu}_{n_1} \qquad n_1 \le n_2}{\Gamma \vdash e : {}_m\tau^{\nu}_{n_2}}\ (\textsc{Sub1})$$

$$\frac{\Gamma \vdash e : {}_m\tau^{\emptyset}_{n}}{\Gamma \vdash e : {}_m\tau^{\dagger}_{n}}\ (\textsc{Sub2})$$

Figure 4: Statics of Pedant

none of the three subexpressions contain newlines, and therefore the entire if statement appears on one line:

```
if b then f () else g ()
```

All three subexpressions must begin with at least one space, in order to separate the expression from the preceding keyword. This is enforced with the +1 on the left component of the type of each subexpression. Rule bool-E2 assumes that the condition has no newlines, but requires the "then" branch to have starting and ending newlines and the "else" branch to have a top newline. As before, all subexpressions must begin with whitespace. Now, however, this results in requiring at least one space of indentation for the two branches, as both are required to begin with a newline.

```
if b then
  f ()
else
  g ()
```

Finally, rule bool-E3 additionally assumes that the condition has starting and ending newlines.

```
if b then f ()
else g ()
```

```
if b then
  f ()
else g ()
```

```
if
b then
  f ()
else g ()
```

Figure 5: Badly formatted expressions that can't typecheck in Pedant.

```
if
  b
then
  f ()
else
  g ()
```

Note that because these are the only three typing rules for "if", none of the abominations in Figure 5 can typecheck.

In all cases, the left column of the if expression is 0, because the expression does not start with whitespace. The width is calculated appropriately from the widths of the subexpressions and keywords. For multiline expressions, we take the maximum width. Note that in bool-E2, for example, both branches are assigned the same width. To make this work, we allow for a form of subtyping (Sub1) which we call *width subtyping*, a name so well-suited to this concept that I won't bother Googling for whether or not it's already used for a different concept. This allows a narrower expression to be assigned a wider type for the purposes of taking the maximum width of several subexpressions. We also allow for another form of subtyping on line breaks (Sub2) in which "none" is considered a subtype of "internal". For this, we will use the similarly well-suited term *depth subtyping*.

At this point in a type systems paper, we would normally define a dynamic semantics for the language, prove progress and preservation and declare success. The astute reader may notice, however, that the type system we have defined fails at even the most basic aspects of being a type system. In fact, not only would preservation be false for any reasonable dynamic semantics, but we can't even prove a reasonable substitution lemma:

$$x : \mathsf{int} \vdash x : {}_0\mathsf{int}_1^{\emptyset}$$

$$[42/x]x = 42$$

$$\cdot \vdash 42 : {}_0\mathsf{int}_2^{\emptyset}$$

At this point, we have several options:

(1) We could admit that this whole idea is a farce and these are entirely syntactic properties that we have no business trying to enforce semantically, and scrap this idea altogether.
(2) We could quietly admit the above but recognize that this whole conference is a farce and decide that it doesn't matter.
(3) We could double down and go to even more ridiculous lengths to try to make this idea at least *seem* reasonable.

For maximum comedic effect, we will, of course, proceed with Option 3.

```
let c = "SIGBOVIK␣2022" in
let how = "really␣really␣really␣" in
let what = "bad␣idea" in
"This␣" ^ c ^ "␣paper␣is␣a␣" ^ how ^ what

let how = "really␣really␣really␣" in
let what = "bad␣idea" in
"This␣" ^ "SIGBOVIK␣2022" ^ "␣paper␣is␣a␣" ^ how ^ what

let what = "bad␣idea" in
"This␣" ^ "SIGBOVIK␣2022" ^ "␣paper␣is␣a␣" ^ "really␣really␣really␣" ^

...
```

Figure 6: A single-step execution of an OCaml expression.

3 THE LANGUAGE PRETTYPRINT

Before attempting to make the ideas of the previous section kinda sorta work, we need a flimsy justification for doing so. Most functional languages are presented with an operational semantics that involves transforming expressions (e.g. applying substitutions, reducing "if" statements when the condition is evaluated), as opposed to imperative languages, which are generally presented as static code with a program counter that captures the runtime control flow. This poses a problem for debuggers for functional languages: while debuggers for imperative languages can easily show the line of source code corresponding to the point of execution and display the values of variables, debuggers for functional languages are generally not able to do something immediately analogous. Part of the problem is that functional languages are generally not actually evaluated in a way that closely resembles the abstract operational semantics presentation: they are either changed heavily during compilation or interpreted using more efficient techniques. However, one could imagine building an educational debugger for a functional language that allows novice functional programmers to single-step programs in a way that follows the formal operational semantics they learned for the language. This then presents the problem of displaying the program at any point during execution. One option would be to maintain the AST of the program as it is transformed by execution, and pretty-print it when needed. However, this would obscure many of the transformations. Statements that had been spread across many lines might now be condensed into one or vice versa, and it would be difficult for novices to follow how the code moves across the screen. It would be preferable for this contrived example if the line breaks in the original code were maintained, so the steps were clear. But then, with no restrictions on input programs, it would be possible for some of these intermediate expressions to be too wide to properly display. Consider the example in Figure 6.

This flimsy justification gives us a new, and actually less ridiculous, interpretation for the types of expressions: the type should describe the maximum width that an expression will have during execution, rather than simply the width of the source expression. This now begins to more closely resemble an *effect* or *type-and-effect* system. As with most effect systems, this requires us to annotate more types. For example, it is now no longer sufficient to describe a function type by its input and output base types, its left column and its width. Effect systems for call-by-value languages would

affect
verb tr.

(1) (of things) to tend toward habitually or naturally.
(2) to assume artificially, pretentiously, or for effect: *to affect a Southern accent.*

Figure 7: Definitions of *affect*, paraphrased from Dictionary.com [1]

generally annotate the function type (or possibly the return type, depending on the desired notation) with the effects that might occur during execution of the function. We could similarly annotate the return type of a function with a left, width and newline description, indicating the behavior of the function as it executes. However, this will depend on the width and line breaks of the substituted arguments. Consider the following function:

```
fun x -> "Is this too long? It depends on " ^ x
```

So we must also annotate the domains of functions with their width and line breaks (it will turn out not to be necessary to annotate the left of domains because of how we design the dynamics). This is not generally the case in effect systems for call-by-value languages because passed arguments will be values, which will by definition have no effects. We observe, however that the width of an expression is not really an effect, but rather an *innate property* of the expression which we must consider for both values and non-value expressions. Because of this, we refer to our novel construction as a *type-and-affect* system (see Figure 7).

In keeping with our new motivation, we will call this language PrettyPrint. The revised statics appear in Figure 8. For the most part, the rules are similar to those for Pedant. In addition to annotating the types of codomains and domains of functions (and the components of product types), we now annotate variables in the context with their width and newlines. The variable rule then assigns a variable x the maximum of $|x|$ and the width from the context. The function application rules ensure that function arguments match the function's expected width and newlines, with the caveat that because of depth subtyping, arguments with no line breaks, i.e., arguments written in one row of text, can be passed to functions expecting multi-row "internal" arguments, allowing us a form of *row polymorphism* (again, no Googling necessary).

Finally, we present the dynamics of PrettyPrint in Figure 9. The dynamics depend on two auxiliary definitions, $\mathit{strip}(e)$ and $\overleftarrow{e}$, defined in Figure 10. The function $\mathit{strip}(e)$ strips leading and trailing whitespace from an expression. The function $\overleftarrow{e}$ removes indentation from e. It is defined in terms of $\mathit{Indentation}(e)$, which calculates the indentation level of e. We remove indentation and trailing and leading whitespace before performing substitution. We must also look past whitespace to perform reductions. Otherwise, the semantics are standard.

We will now state, without any attempt at proof, several facts about the correctness of these operations that are probably at least close to true.

Lemma 1. *If* $\Gamma \vdash e : {}_m\tau\frac{\nu}{n}$ *then* $\Gamma \vdash \overleftarrow{e} : {}_0\tau\frac{\nu}{n-\mathit{Indentation}(e)}$.

Lemma 2. *If* $\Gamma \vdash e : {}_m\tau\frac{\nu}{n}$ *then* $\Gamma \vdash \mathit{strip}(e) : {}_m\tau^{\dagger}_{n}$. *If* $\Gamma \vdash e : {}_m\tau^{\emptyset}_{n}$ *then* $\Gamma \vdash \mathit{strip}(e) : {}_m\tau^{\emptyset}_{n}$.

Lemma 3 (Substitution). *If* $\Gamma, v : \tau_1\frac{\nu_1}{k} \vdash e : {}_m\tau_2\frac{\nu_2}{n}$ *and* $\Gamma \vdash v : {}_j\tau_1\frac{\nu_1}{k}$, *then* $\Gamma \vdash e[\mathit{strip}(\overleftarrow{v})/x] : {}_m\tau_2\frac{\nu_2}{n}$.

Finally, we can state and not attempt to prove type safety.

Theorem 1 (Preservation). *If* $\bullet \vdash e : {}_m\tau\frac{\nu}{n}$ *and* $e \mapsto e'$ *then* $\bullet \vdash e' : {}_m\tau\frac{\nu}{n}$.

Of course, the canonical forms lemma now becomes interesting because irreducible values may have leading or trailing whitespace.

Lemma 4 (Canonical Forms).

(1) *If* $\bullet \vdash v : {}_m\mathsf{int}\frac{\nu}{n}$, *then* $\mathit{strip}(v) = \overline{n}$ *for some* n.
(2) *If* $\bullet \vdash v : {}_m\mathsf{bool}\frac{\nu}{n}$, *then* $\mathit{strip}(v) = \mathsf{true}$ *or* $\mathit{strip}(v) = \mathsf{false}$.
(3) *If* $\bullet \vdash v : {}_m\tau_1 \to \tau_2\frac{\nu}{n}$, *then* $\mathit{strip}(v) = \mathsf{fun}\ x\ {-}{>}\ e$.
(4) *If* $\bullet \vdash v : {}_{l\,m}\tau_1\frac{\nu_1}{n} \times {}_j\tau_2\frac{\nu_2}{k}\frac{\nu}{w}$ *then* $\mathit{strip}(v) = (v_1, v_2)$.

Theorem 2 (Progress). *If* $\bullet \vdash e : {}_m\tau\frac{\nu}{n}$ *then* e *is a value or there exists* e' *such that* $e \mapsto e'$.

4 IMPLEMENTATION

Yes, you read that section header correctly. This paper is not a joke. Well, it is. But it's a joke to which the author is deeply, deeply committed for reasons that are not clear in the slightest. So yeah, we implemented a mostly[2]-working parser, type checker and single-step interpreter for PrettyPrint.

Type inference is about as awful as you'd expect. The type system is too weird for standard unification algorithms, so instead, the implementation traverses the program and generates a set of constraints on the width, left and newline behavior of each expression. At the end, a constraint is added restricting the overall width of the expression to be, of course, 80. We then solve these constraints using Z3 [5]. Note that the constraints on the width and left form an integer linear program (ILP). We leave it to future work to determine whether arbitrary ILPs can be encoded as PrettyPrint programs, making typechecking NP-complete.

After the program is rejected, revised, rejected again, and revised a few more times, and finally typechecked, the completed program is displayed and the user given the option to step the program one step at a time or to completion, as shown in Figure 11. This paper has already gone on too long, so rather than further bore you with details of the implementation, we'll give a few observations about our experience doing the implementation in a nice, easy-to-digest list:

- As the name of the language suggests, pretty-printing, which, frankly, I've always found to be the most difficult part of language implementation, is now trivial.
- Lexing, typically the most trivial part of language implementation, is now even more trivial as you don't even need to figure out how to skip over whitespace.

[2] well, somewhat

$$\frac{}{\Gamma, x : \tau \overset{\nu}{n} \vdash x : {}_0\tau \overset{\nu}{\max(n, |x|)}} \text{(Var)} \qquad \frac{}{\Gamma \vdash \overline{n} : {}_0\text{int} \overset{\emptyset}{|\overline{n}|}} \text{(int-I)} \qquad \frac{}{\Gamma \vdash \text{true} : {}_0\text{bool} \overset{\emptyset}{4}} \text{(bool-I1)} \qquad \frac{}{\Gamma \vdash \text{false} : {}_0\text{bool} \overset{\emptyset}{5}} \text{(bool-I2)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : {}_{m+1}\tau_2 \overset{\emptyset}{n}}{\Gamma \vdash \text{fun␣}x\text{␣->}\ e : {}_0\tau_1 \to \tau_2 \overset{\emptyset}{4+|x|+4+m+n}} \text{(→-I1)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : {}_{m+1}\tau_2 \overset{\top}{n}}{\Gamma \vdash \text{fun␣}x\text{␣->}\ e : {}_0\tau_1 \to \tau_2 \overset{\dagger}{\max(m+1+n, 4+|x|+4)}} \text{(→-I2)}$$

$$\frac{\Gamma \vdash e_1 : {}_m\tau_1 \overset{\emptyset}{k} \to \tau_2 \overset{\emptyset}{n} \qquad \Gamma \vdash e_2 : {}_j\tau_1 \overset{\emptyset}{k}}{\Gamma \vdash e_1\text{␣}e_2 : {}_m\tau_2 \overset{\nu}{n+1+j+k}} \text{(→-E1)} \qquad \frac{\Gamma \vdash e_1 : {}_m\tau_1 \overset{\dagger}{n} \to \tau_2 \overset{\bot}{n+1} \qquad \Gamma \vdash e_2 : {}_{m+1}\tau_1 \overset{\dagger}{n}}{\Gamma \vdash e_1\text{␣}e_2 : {}_m\tau_2 \overset{\dagger}{n+1}} \text{(→-E2)}$$

$$\frac{\Gamma \vdash e_1 : {}_m\tau_1 \overset{\dagger}{n} \to \tau_2 \overset{\dagger}{n+1} \qquad \Gamma \vdash e_2 : {}_{m+1}\tau_1 \overset{\top}{n}}{\Gamma \vdash e_1\text{␣}e_2 : {}_m\tau_2 \overset{\dagger}{n+1}} \text{(→-E3)} \qquad \frac{\Gamma \vdash e_1 : {}_m\tau_1 \overset{\emptyset}{n} \qquad \Gamma \vdash e_2 : {}_j\tau_2 \overset{\emptyset}{k}}{\Gamma \vdash (e_1, e_2) : {}_0\tau_1 \times \tau_2 \overset{\emptyset}{m+n+j+k+3}} \text{(×-I1)} \qquad \frac{\Gamma \vdash e_1 : {}_m\tau_1 \overset{\dagger}{n} \qquad \Gamma \vdash e_2 : {}_{m+1}\tau_2 \overset{\top}{n}}{\Gamma \vdash (e_1, e_2) : {}_0\tau_1 \times \tau_2 \overset{\dagger}{m+n+1}} \text{(×-I2)}$$

$$\frac{\Gamma \vdash e_1 : {}_m\left(\tau_1 \overset{\nu_1}{w_1} \times \tau_2 \overset{\nu_2}{w_2}\right) \overset{\emptyset}{n} \qquad \Gamma, x : \tau_1 \overset{\nu_1}{w_1}, y : \tau_2 \overset{\nu_2}{w_2} \vdash e_2 : {}_{j+1}\tau' \overset{\emptyset}{k}}{\Gamma \vdash \text{let } (x,\text{␣}y) = e_1 \text{ in } e_2 : {}_0\tau' \overset{\emptyset}{5+|x|+2+|y|+3+m+n+3+j+1+k}} \text{(×-E1)}$$

$$\frac{\Gamma \vdash e_1 : {}_m\left(\tau_1 \overset{\nu_1}{w_1} \times \tau_2 \overset{\nu_2}{w_2}\right) \overset{\emptyset}{n} \qquad \Gamma, x : \tau_1 \overset{\nu_1}{w_1}, y : \tau_2 \overset{\nu_2}{w_2} \vdash e_2 : {}_{j+1}\tau' \overset{\text{I}}{k}}{\Gamma \vdash \text{let } (x,\text{␣}y) = e_1 \text{ in } e_2 : {}_0\tau' \overset{\dagger}{\max(5+|x|+2+|y|+3+m+n+3, j+1+k)}} \text{(×-E2)}$$

$$\frac{\Gamma \vdash e_1 : {}_{m+1}\left(\tau_1 \overset{\nu_1}{w_1} \times \tau_2 \overset{\nu_2}{w_2}\right) \overset{\text{I}}{n} \qquad \Gamma, x : \tau_1 \overset{\nu_1}{w_1}, y : \tau_2 \overset{\nu_2}{w_2} \vdash e_2 : {}_{m+1}\tau' \overset{\text{I}}{n}}{\Gamma \vdash \text{let } (x,\text{␣}y) = e_1 \text{ in } e_2 : {}_0\tau' \overset{\dagger}{\max(5+|x|+2+|y|+3, m+1+n)}} \text{(×-E3)}$$

$$\frac{\Gamma \vdash e_1 : {}_{m+1}\text{bool} \overset{\emptyset}{n} \qquad \Gamma \vdash e_2 : {}_{j+1}\tau \overset{\emptyset}{k} \qquad \Gamma \vdash e_3 : {}_{i+1}\tau \overset{\emptyset}{l}}{\Gamma \vdash \text{if } e_1\text{␣then } e_2\text{␣else } e_3 : {}_0\tau \overset{\emptyset}{2+m+1+n+5+j+1+k+5+i+1+l}} \text{(bool-E1)} \qquad \frac{\Gamma \vdash e_1 : {}_{m+1}\text{bool} \overset{\emptyset}{n} \qquad \Gamma \vdash e_2 : {}_{j+1}\tau \overset{\text{I}}{k} \qquad \Gamma \vdash e_3 : {}_{j+1}\tau \overset{\top}{k}}{\Gamma \vdash \text{if } e_1\text{␣then } e_2\text{␣else } e_3 : {}_0\tau \overset{\dagger}{\max(2+m+1+n, j+1+k)}} \text{(bool-E2)}$$

$$\frac{\Gamma \vdash e_1 : {}_{m+1}\text{bool} \overset{\text{I}}{n} \qquad \Gamma \vdash e_2 : {}_{m+1}\tau \overset{\text{I}}{n} \qquad \Gamma \vdash e_3 : {}_{m+1}\tau \overset{\dagger}{n}}{\Gamma \vdash \text{if } e_1\text{␣then } e_2\text{␣else } e_3 : {}_0\tau \overset{\dagger}{\max(4, m+1+n)}} \text{(bool-E3)} \qquad \frac{\Gamma \vdash e_1 : {}_m\tau_1 \overset{\emptyset}{n} \qquad \Gamma, x : \tau_1 \overset{\emptyset}{n} \vdash e_2 : {}_{j+1}\tau_2 \overset{\emptyset}{k}}{\Gamma \vdash \text{let␣}x\text{␣} = e_1\text{␣in } e_2 : {}_0\tau' \overset{\emptyset}{4+|x|+2+m+n+3+j+1+k}} \text{(Let1)}$$

$$\frac{\Gamma \vdash e_1 : {}_m\tau_1 \overset{\emptyset}{n} \qquad \Gamma, x : \tau_1 \overset{\emptyset}{n} \vdash e_2 : {}_{j+1}\tau_2 \overset{\text{I}}{k}}{\Gamma \vdash \text{let␣}x\text{␣} = e_1\text{␣in } e_2 : {}_0\tau' \overset{\dagger}{\max(4+|x|+2+m+n+3, j+1+k)}} \text{(Let2)} \qquad \frac{\Gamma \vdash e_1 : {}_{m+1}\tau_1 \overset{\text{I}}{n} \qquad \Gamma, x : \tau_1 \overset{\text{I}}{n} \vdash e_2 : {}_{m+1}\tau_2 \overset{\text{I}}{n}}{\Gamma \vdash \text{let␣}x\text{␣} = e_1\text{␣in } e_2 : {}_0\tau' \overset{\dagger}{\max(4+|x|+2, m+1+n)}} \text{(Let3)}$$

$$\frac{\Gamma \vdash e : {}_m\tau \overset{\emptyset}{n}}{\Gamma \vdash \text{␣}e : {}_{m+1}\tau \overset{\emptyset}{n}} \text{(Space1)} \qquad \frac{\Gamma \vdash e : {}_m\tau \overset{\emptyset}{n}}{\Gamma \vdash e\text{␣} : {}_m\tau \overset{\emptyset}{n+1}} \text{(Space2)} \qquad \frac{\Gamma \vdash e : {}_m\tau \overset{\dagger}{n}}{\Gamma \vdash \backslash\text{n}e : {}_m\tau \overset{\top}{n}} \text{(Newline1)} \qquad \frac{\Gamma \vdash e : {}_m\tau \overset{\bot}{n}}{\Gamma \vdash \backslash\text{n}e : {}_m\tau \overset{\text{I}}{n}} \text{(Newline2)}$$

$$\frac{\Gamma \vdash e : {}_m\tau \overset{\dagger}{n}}{\Gamma \vdash e\backslash\text{n} : {}_m\tau \overset{\bot}{n}} \text{(Newline3)} \qquad \frac{\Gamma \vdash e : {}_m\tau \overset{\top}{n}}{\Gamma \vdash e\backslash\text{n} : {}_m\tau \overset{\text{I}}{n}} \text{(Newline2)} \qquad \frac{\Gamma \vdash e : {}_m\tau \overset{\nu}{n_1} \qquad n_1 \le n_2}{\Gamma \vdash e : {}_m\tau \overset{\nu}{n_2}} \text{(Sub1)} \qquad \frac{\Gamma \vdash e : {}_m\tau \overset{\emptyset}{n}}{\Gamma \vdash e : {}_m\tau \overset{\dagger}{n}} \text{(Sub2)}$$

Figure 8: Statics of PrettyPrint

- Producing reasonable type error messages, possibly the second most difficult part of language implementation behind pretty printing, is actually a little less painful than you might expect for this language. We associate semantic and position information with each constraint passed to Z3. If the constraints are unsatisfiable, we look up this information for

$$\frac{e_1 \mapsto e_1'}{e_1 \_ e_2 \mapsto e_1' \_ e_2} \qquad \frac{e_2' \mapsto e_2'}{v_1 \_ e_2 \mapsto v_1 \_ e_2'} \qquad \frac{strip(v_1) = \mathsf{fun}\_x\_{-}\!> e}{v_1 \_ v_2 \mapsto e[strip(\overleftarrow{v_2})/x]} \qquad \frac{e_1 \mapsto e_1'}{(e_1, e_2) \mapsto (e_1', e_2)} \qquad \frac{e_2' \mapsto e_2'}{(v_1, e_2) \mapsto (v_1, e_2')}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{let}\ (x, \_y) = e_1\ \mathsf{in}\ e_2 \mapsto \mathsf{let}\ (x, \_y) = e_1'\ \mathsf{in}\ e_2} \qquad \frac{}{\mathsf{let}\ (x, \_y) = (v_1, v_2)\ \mathsf{in}\ e_2 \mapsto e_2[strip(\overleftarrow{v_1})/x][strip(\overleftarrow{v_2})/y]}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{let}\_x\_ = e_1 \_ \mathsf{in}\ e_2 \mapsto \mathsf{let}\_x\_ = e_1' \_ \mathsf{in}\ e_2} \qquad \frac{}{\mathsf{let}\_x\_ = v \_ \mathsf{in}\ e_2 \mapsto e_2[strip(\overleftarrow{v})/x]} \qquad \frac{e_1 \mapsto e_1'}{\mathsf{if}\ e_1 \_ \mathsf{then}\ e_2 \_ \mathsf{else}\ e_3 \mapsto \mathsf{if}\ e_1' \_ \mathsf{then}\ e_2 \_ \mathsf{else}\ e_3}$$

$$\frac{strip(v) = \mathsf{true}}{\mathsf{if}\ v \_ \mathsf{then}\ e_2 \_ \mathsf{else}\ e_3 \mapsto e_2} \qquad \frac{strip(v) = \mathsf{false}}{\mathsf{if}\ v \_ \mathsf{then}\ e_2 \_ \mathsf{else}\ e_3 \mapsto e_3} \qquad \frac{e \mapsto e'}{\_ e \mapsto \_ e'} \qquad \frac{e \mapsto e'}{e \_ \mapsto e' \_} \qquad \frac{e \mapsto e'}{\backslash \mathsf{n} e \mapsto \backslash \mathsf{n} e'} \qquad \frac{e \mapsto e'}{e \backslash \mathsf{n} \mapsto e' \backslash \mathsf{n}}$$

Figure 9: Dynamics of PrettyPrint

each constraint in the unsat core and convert it to halfway-decent error messages.

5 CONCLUSION

I first had the idea for this paper right after SIGBOVIK 2020 (right after SIGBOVIK is, of course, when I have all of my best SIGBOVIK paper ideas). At least, I thought, I had a year to make it actually work out. Of course, I forgot all about this until January 2021 and figured I still had enough time to throw something together. Unfortunately, making this really work, which I was determined to do, took quite a bit more time than that. So, I missed the deadline and figured, well, now I have *another* whole year.

And forgot about it until January 2022. But this time, I managed to scrape it together and submit.

Two years, two core calculi and way too much implementation effort later, we have a dumb language that enforces pedantic white-space constraints during type checking. Was it worth it? That's for you to decide.

REFERENCES

[1] [n. d.]. Affect. https://www.dictionary.com/browse/affect. Accessed: 3/21/2021.
[2] [n. d.]. Style Guide for C. https://cs50.readthedocs.io/style/c/. Accessed: 3/21/2021.
[3] [n. d.]. Whitespace .NET. Accessed: 3/24/2022.
[4] L.W. Cannon, R.A. Elliott, L.W. Kirchhoff, et al. [n. d.]. Recommended C Style and Coding Standards.
[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

$$
\begin{array}{lcl}
\mathit{strip}(\mathsf{fun}\,x\,{-}{>}\,e) & = & \mathsf{fun}\,x\,{-}{>}\,\mathit{rstrip}(e) \\
\mathit{strip}(e_1\,e_2) & = & \mathit{lstrip}(e_1)\,\mathit{rstrip}(e_2) \\
\mathit{strip}(\mathsf{let}\,(x,\,y) = e_1\,\mathsf{in}\,e_2) & = & \mathsf{let}\,(x,\,y) = e_1\,\mathsf{in}\,\mathit{rstrip}(e_2) \\
\mathit{strip}(\mathsf{let}\,x = e_1\,\mathsf{in}\,e_2) & = & \mathsf{let}\,x = e_1\,\mathsf{in}\,\mathit{rstrip}(e_2) \\
\mathit{strip}(\mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3) & = & \mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,\mathit{rstrip}(e_3) \\
\mathit{strip}(\,e) & = & \mathit{strip}(e) \\
\mathit{strip}(e\,) & = & \mathit{strip}(e) \\
\mathit{strip}(\backslash\mathsf{n}e) & = & \mathit{strip}(e) \\
\mathit{strip}(e\backslash\mathsf{n}) & = & \mathit{strip}(e)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{lstrip}(e_1\,e_2) & = & \mathit{lstrip}(e_1)\,e_2 \\
\mathit{lstrip}(\,e) & = & \mathit{lstrip}(e) \\
\mathit{lstrip}(e\,) & = & \mathit{lstrip}(e)\, \\
\mathit{lstrip}(\backslash\mathsf{n}e) & = & \mathit{lstrip}(e) \\
\mathit{lstrip}(e\backslash\mathsf{n}) & = & \mathit{lstrip}(e)\backslash\mathsf{n}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{rstrip}(\mathsf{fun}\,x\,{-}{>}\,e) & = & \mathsf{fun}\,x\,{-}{>}\,\mathit{rstrip}(e) \\
\mathit{rstrip}(e_1\,e_2) & = & e_1\,\mathit{rstrip}(e_2) \\
\mathit{rstrip}(\mathsf{let}\,(x,\,y) = e_1\,\mathsf{in}\,e_2) & = & \mathsf{let}\,(x,\,y) = e_1\,\mathsf{in}\,\mathit{rstrip}(e_2) \\
\mathit{rstrip}(\mathsf{let}\,x = e_1\,\mathsf{in}\,e_2) & = & \mathsf{let}\,x = e_1\,\mathsf{in}\,\mathit{rstrip}(e_2) \\
\mathit{rstrip}(\mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3) & = & \mathsf{if}\,e_1\,\mathsf{then}\,e_2\,\mathsf{else}\,\mathit{rstrip}(e_3) \\
\mathit{rstrip}(\,e) & = & \,\mathit{rstrip}(e) \\
\mathit{rstrip}(e\,) & = & \mathit{rstrip}(e) \\
\mathit{rstrip}(\backslash\mathsf{n}e) & = & \backslash\mathsf{n}\mathit{rstrip}(e) \\
\mathit{rstrip}(e\backslash\mathsf{n}) & = & \mathit{rstrip}(e)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Indentation}(v) & = & 0 \\
\mathit{Indentation}(\,e) & = & 1 + \mathit{Indentation}(e) \\
\mathit{Indentation}(e\,) & = & \mathit{Indentation}(e) \\
\mathit{Indentation}(\backslash\mathsf{n}e) & = & \mathit{Indentation}(e) \\
\mathit{Indentation}(e\backslash\mathsf{n}) & = & \mathit{Indentation}(e)
\end{array}
$$

$$
\begin{array}{lcl}
\overleftarrow{\,e}^{(m+1,\,n)} & = & \overleftarrow{e}^{(m,\,n)} \\
\overleftarrow{e\,}^{(m,\,n)} & = & \overleftarrow{e}^{(m,\,n)}\, \\
\overleftarrow{\backslash\mathsf{n}e}^{(m,\,n)} & = & \backslash\mathsf{n}\overleftarrow{e}^{(n,\,n)} \\
\overleftarrow{e\backslash\mathsf{n}}^{(m,\,n)} & = & \overleftarrow{e}^{(m,\,n)}\backslash\mathsf{n} \\
\overleftarrow{e} & = & \overleftarrow{e}^{(\mathit{Indentation}(e),\,\mathit{Indentation}(e))}
\end{array}
$$

Figure 10: Auxiliary definitions for dynamics.

```
$ ./pdb prog.prp
Loading prog.prp...
Type checking...
Done.
--------
let x = "I can't believe" in
  let y = "this works" in
    (x, y)
--------
s
--------

  let y = "this works" in
    ("I can't believe", y)
--------
s
--------


    ("I can't believe", "this works")
--------
s
Execution is complete.
--------


    ("I can't believe", "this works")
--------
q
```

Figure 11: Example run of the PrettyPrint interpreter.