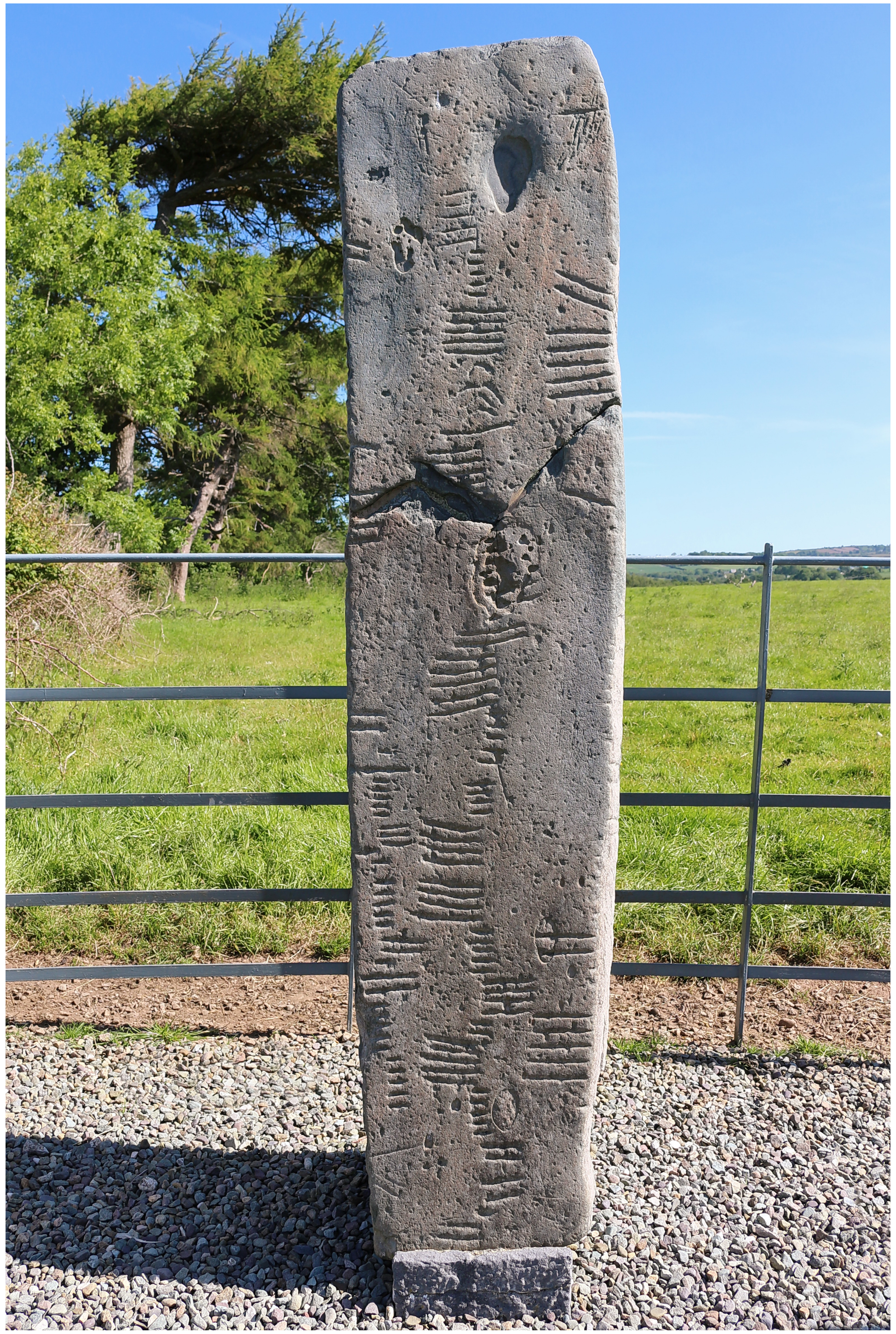
Is There a Future for a Formally Specified Haskell Report?

March 9th 2026 @ PLaS Group Kent

David Binder

University of Kent


(c) Robert Linsdell CC-BY

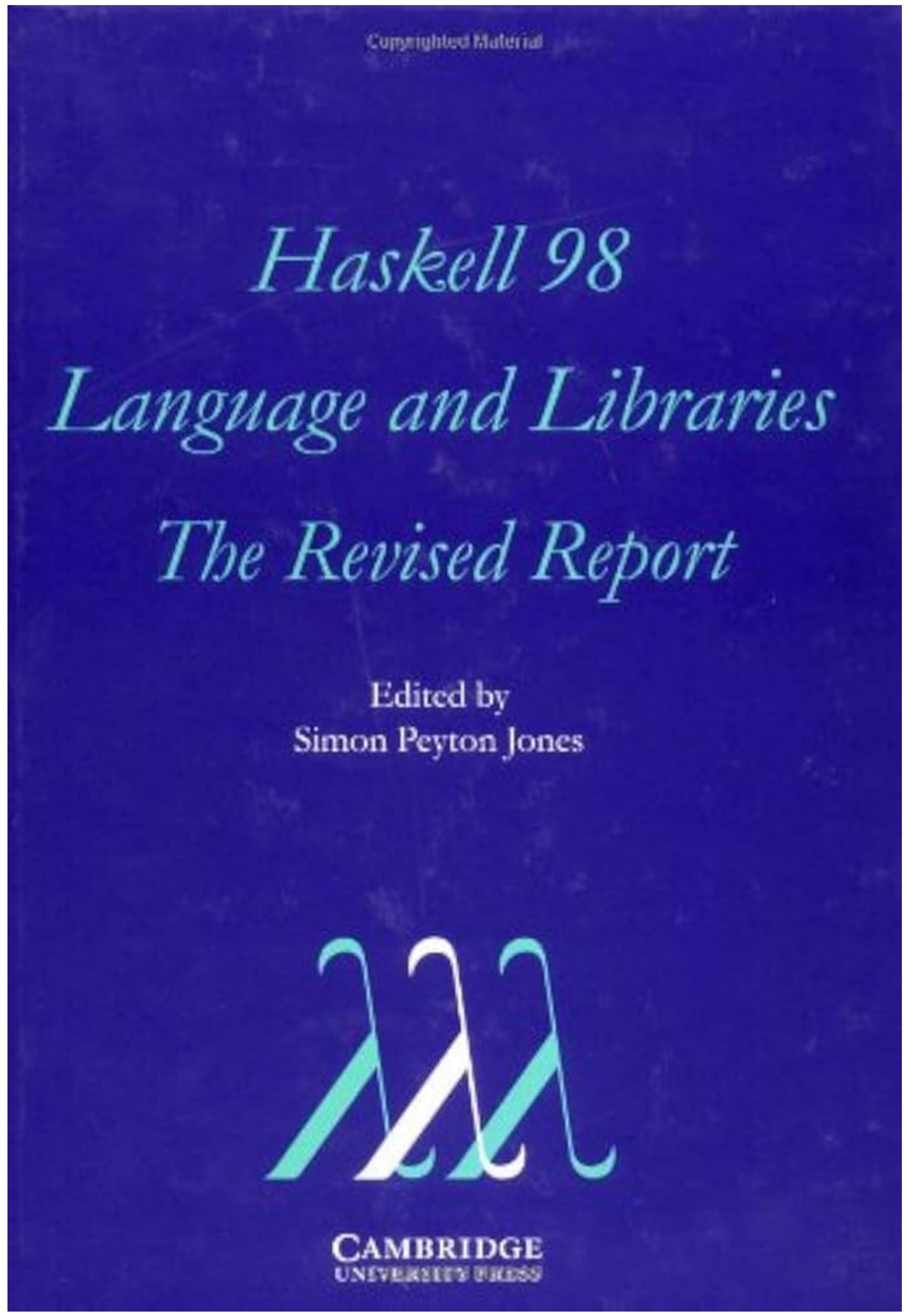
Ogham

᚛ᚑᚌᚐᚋ᚜

Do Languages Allow Ogham Whitespace?

Python	✓
Rust	✗
C	(Depends on GCC vs Clang)
Haskell	✓

What Does the Haskell Report Say


Copyrighted Material

Haskell 98
Language and Libraries
The Revised Report

Edited by
Simon Peyton Jones

CAMBRIDGE
UNIVERSITY PRESS

whitespace	$\rightarrow$	$\mathit{whitestuff}\ \{\mathit{whitestuff}\}$
whitestuff	$\rightarrow$	$\mathit{whitechar} \mid \mathit{comment} \mid \mathit{ncomment}$
whitechar	$\rightarrow$	$\mathit{newline} \mid \mathit{vertab} \mid \mathit{space} \mid \mathit{tab} \mid \mathit{uniWhite}$
newline	$\rightarrow$	$\mathit{return}\ \mathit{linefeed} \mid \mathit{return} \mid \mathit{linefeed} \mid \mathit{formfeed}$
return	$\rightarrow$	a carriage return
linefeed	$\rightarrow$	a line feed
vertab	$\rightarrow$	a vertical tab
formfeed	$\rightarrow$	a form feed
space	$\rightarrow$	a space
tab	$\rightarrow$	a horizontal tab
uniWhite	$\rightarrow$	any Unicode character defined as whitespace

How Does the Haskell Report Look Like

10.1 Notational Conventions

These notational conventions are used for presenting syntax:

[*pattern*]	optional
{*pattern*}	zero or more repetitions
(*pattern*)	grouping
$pat_1 \mid pat_2$	choice
$pat_{\langle pat' \rangle}$	difference—elements generated by *pat* except those generated by *pat′*
`fibonacci`	terminal syntax in typewriter font

BNF-like syntax is used throughout, with productions having the form:

$$nonterm \quad \rightarrow \quad alt_1 \mid alt_2 \mid \dots \mid alt_n$$

3.14 Do Expressions

```
lexp   →  do { stmts }                            (do expression)
stmts  →  stmt1 ... stmtn exp [;]                 (n ≥ 0)
stmt   →  exp ;
       |  pat <- exp ;
       |  let decls ;
       |  ;                                       (empty statement)
```

Translation: Do expressions satisfy these identities, which may be used as a translation into the kernel, after eliminating empty *stmts*:

```
do {e}                  =  e
do {e;stmts}            =  e >> do {stmts}
do {p <- e; stmts}      =  let ok p = do {stmts}
                               ok _ = fail "..."
                             in e >>= ok
do {let decls; stmts}   =  let decls in do {stmts}
```

The ellipsis `"..."` stands for a compiler-generated error message, passed to `fail`, preferably giving some indication of the location of the pattern-match failure; the functions `>>`, `>>=`, and `fail` are operations in the class `Monad`, as defined in the Prelude; and `ok` is a fresh identifier.

How Does the Haskell Report Look Like

4.6 Kind Inference

This section describes the rules that are used to perform *kind inference*, i.e. to calculate a suitable kind for each type constructor and class appearing in a given program.

The first step in the kind inference process is to arrange the set of datatype, synonym, and class definitions into dependency groups. This can be achieved in much the same way as the dependency analysis for value declarations that was described in Section 4.5. For example, the following program fragment includes the definition of a datatype constructor `D`, a synonym `S` and a class `C`, all of which would be included in the same dependency group:

```
data C a => D a = Foo (S a)
type S a = [D a]
class C a where
    bar :: a -> D a -> Bool
```

The kinds of variables, constructors, and classes within each group are determined using standard techniques of type inference and kind-preserving unification [8]. For example, in the definitions above, the parameter `a` appears as an argument of the function constructor `(->)` in the type of `bar` and hence must have kind $*$. It follows that both `D` and `S` must have kind $* \to *$ and that every instance of class `C` must have kind $*$.

It is possible that some parts of an inferred kind may not be fully determined by the corresponding definitions; in such cases, a default of $*$ is assumed. For example, we could assume an arbitrary kind κ for the `a` parameter in each of the following examples:

```
data App f a = A (f a)
data Tree a  = Leaf | Fork (Tree a) (Tree a)
```

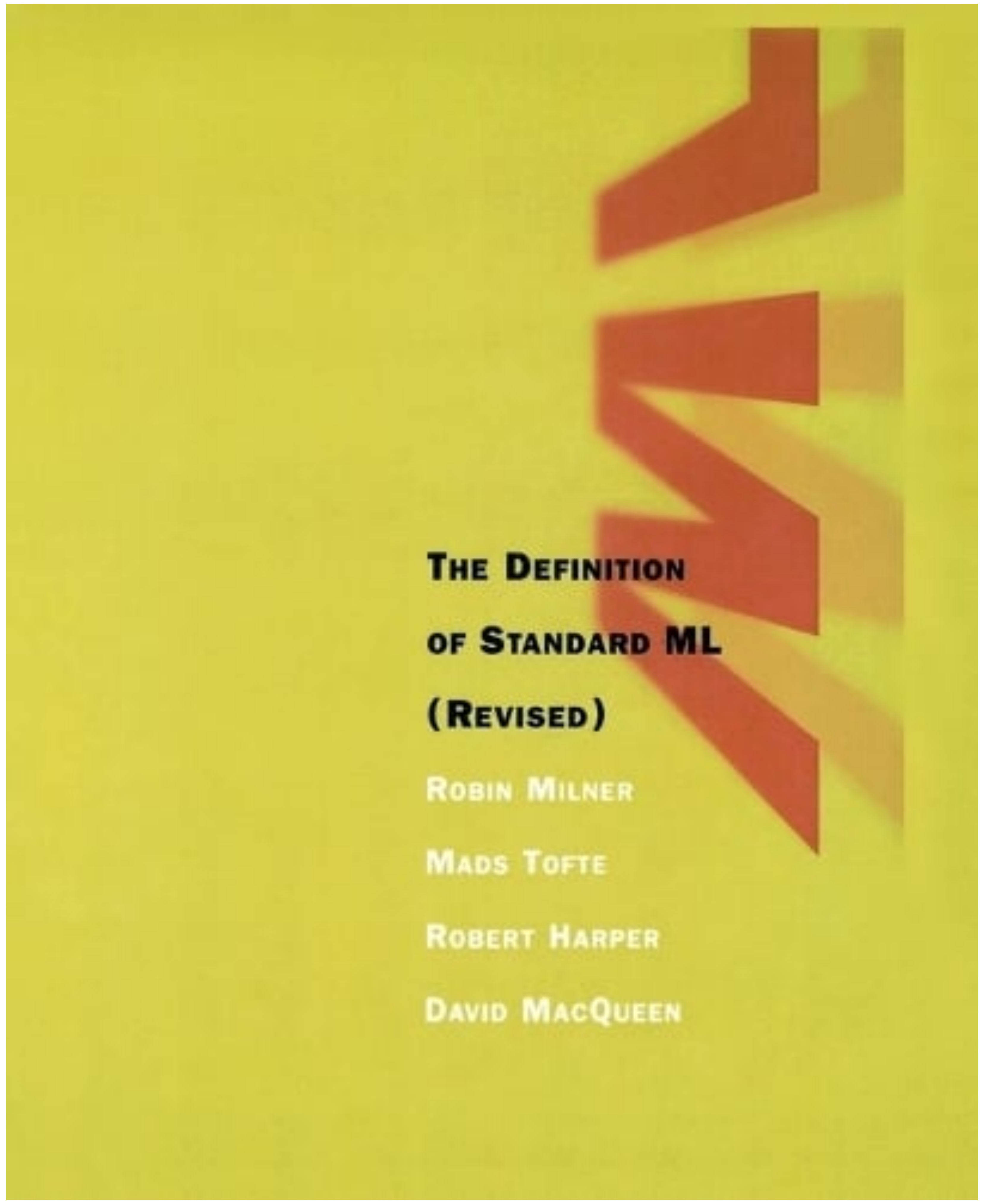
This would give kinds $(\kappa \to *) \to \kappa \to *$ and $\kappa \to *$ for `App` and `Tree`, respectively, for any kind κ, and would require an extension to allow polymorphic kinds. Instead, using the default binding $\kappa = *$, the actual kinds for these two constructors are $(* \to *) \to * \to *$ and $* \to *$, respectively.

Defaults are applied to each dependency group without consideration of the ways in which particular type constructor constants or classes are used in later dependency groups or elsewhere in the program. For example, adding the following definition to those above does not influence the kind inferred for `Tree` (by changing it to $(* \to *) \to *$, for instance), and instead generates a static error because the kind of `[]`, $* \to *$, does not match the kind $*$ that is expected for an argument of `Tree`:

```
type FunnyTree = Tree []     -- invalid
```

This is important because it ensures that each constructor and class are used consistently with the same kind whenever they are in scope.

Definition of Standard ML



Declarations $\boxed{C \vdash dec \Rightarrow E}$

$$\frac{\begin{array}{l} U = \mathrm{tyvars}(tyvarseq) \\ C + U \vdash valbind \Rightarrow VE \qquad VE' = \mathrm{Clos}_{C,valbind} VE \qquad U \cap \mathrm{tyvars}\, VE' = \emptyset \end{array}}{C \vdash \texttt{val}\ \ tyvarseq\ \ valbind \Rightarrow VE' \text{ in Env}} \tag{15}$$

$$\frac{C \vdash typbind \Rightarrow TE}{C \vdash \texttt{type}\ typbind \Rightarrow TE \text{ in Env}} \tag{16}$$

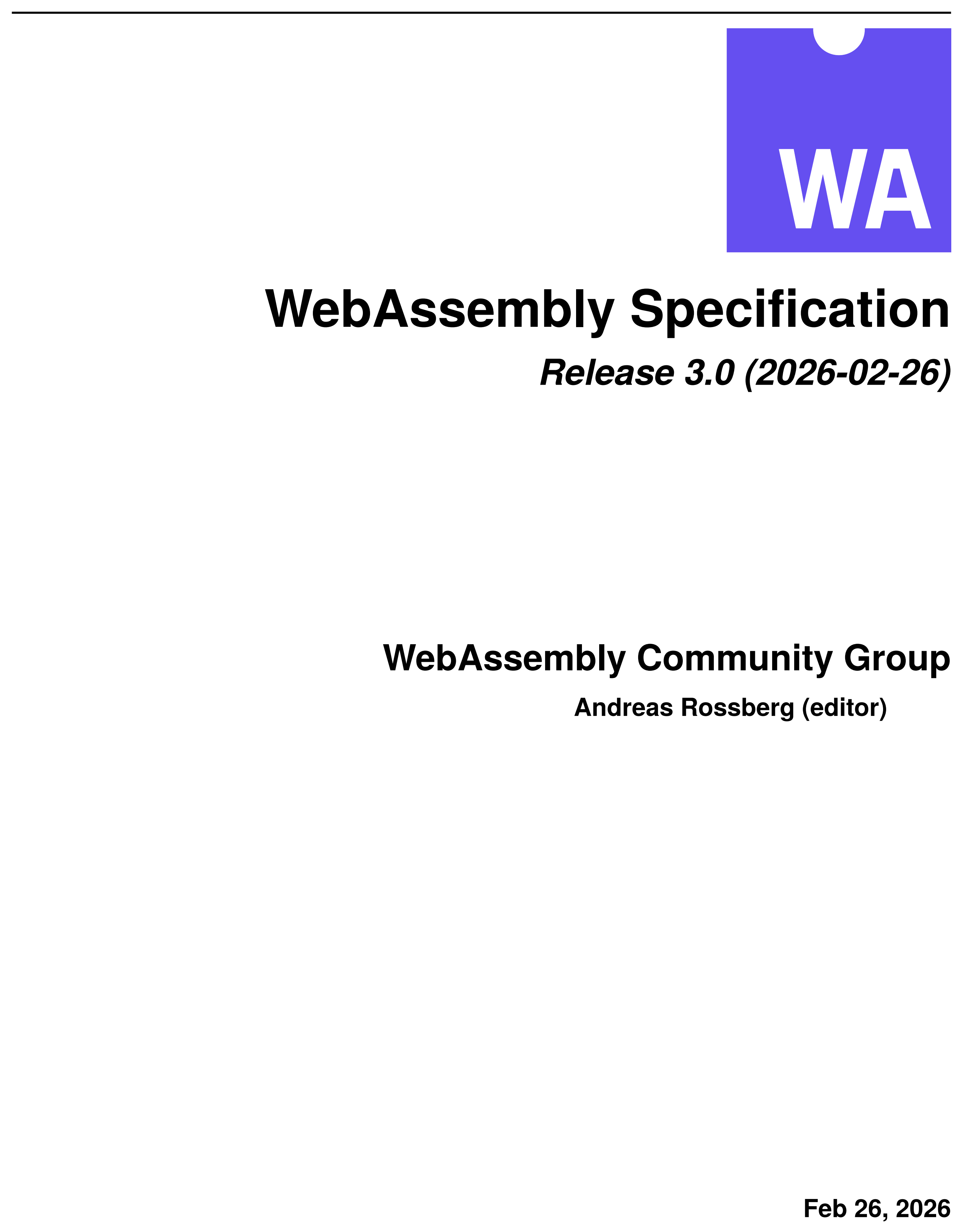
$$\frac{\begin{array}{c} C \oplus TE \vdash datbind \Rightarrow VE, TE \qquad \forall (t, VE') \in \mathrm{Ran}\, TE,\ \ t \notin (T \text{ of } C) \\ TE \text{ maximises equality} \end{array}}{C \vdash \texttt{datatype}\ datbind \Rightarrow (VE, TE) \text{ in Env}} \tag{17}$$

$$\frac{C(longtycon) = (\theta, VE) \qquad TE = \{tycon \mapsto (\theta, VE)\}}{C \vdash \texttt{datatype}\ tycon\ \texttt{-=-}\ \texttt{datatype}\ longtycon \Rightarrow (VE, TE) \text{ in Env}} \tag{18}$$

$$\frac{\begin{array}{ll} C \oplus TE \vdash datbind \Rightarrow VE, TE & \forall (t, VE') \in \mathrm{Ran}\, TE,\ \ t \notin (T \text{ of } C) \\ C \oplus (VE, TE) \vdash dec \Rightarrow E & TE \text{ maximises equality} \end{array}}{C \vdash \texttt{abstype}\ datbind\ \texttt{with}\ dec\ \texttt{end} \Rightarrow \mathrm{Abs}(TE, E)} \tag{19}$$

$$\frac{C \vdash exbind \Rightarrow VE}{C \vdash \texttt{exception}\ exbind \Rightarrow VE \text{ in Env}} \tag{20}$$

Web Assembly Specification



WebAssembly Specification

Release 3.0 (2026-02-26)

WebAssembly Community Group

Andreas Rossberg (editor)

Feb 26, 2026

3.2.11 Recursive Types

Recursive types are validated with respect to the first type index defined by the recursive group.

$\mathsf{rec}\ \mathit{subtype}^*$

The recursive type $(\mathsf{rec}\ \mathit{subtype}^*)$ is valid for the type index x if:

- Either:
  - The sub type sequence $\mathit{subtype}^*$ is empty.
- Or:
  - The sub type sequence $\mathit{subtype}^*$ is of the form $\mathit{subtype}_1\ \mathit{subtype}'^*$.
  - The sub type $\mathit{subtype}_1$ is valid for the type index x.
  - The recursive type $(\mathsf{rec}\ \mathit{subtype}'^*)$ is valid for the type index $x + 1$.

$$\frac{}{C \vdash \mathsf{rec}\ \epsilon : \mathsf{ok}(x)} \qquad \frac{C \vdash \mathit{subtype}_1 : \mathsf{ok}(x) \qquad C \vdash \mathsf{rec}\ \mathit{subtype}^* : \mathsf{ok}(x+1)}{C \vdash \mathsf{rec}\ (\mathit{subtype}_1\ \mathit{subtype}^*) : \mathsf{ok}(x)}$$

Is a Mechanized Haskell Report Possible?

What about a verified compiler?

- Haskell is the language I am most familiar with
- We already have the Haskell 2010 language report
- Many papers formalize specific aspects of Haskell 98 and 2010
- A complete declarative type system for Haskell 98 exists by K-F Faxén
- Mechanizing specifications **(not proofs!)** scales well

The Problem

(c) Charles J. Sharp CC-BY

(c) Wikipedia user: Montrealais CC-BY

Identify Achievable Goals

- Most mechanizations at PL conferences have the goal of proving metatheory
- The goal of this project is to have a machine-checked specification
- Rigorous specification of a realistic programming language requires ingenuity
- **Project 1:** Implement a verified pre-frontend for Haskell 2010
- **Project 2:** Mechanize a formal specification of the declarative type system of Haskell 2010

Project 1: Verified Pre-Frontend

What is a Pre-Frontend

- Computation of Indentation Levels
- Lexical Syntax
- Context Free Syntax
- Renaming and Semantics of the Module System
- Resolving of Operator Fixities
- Computation of Mutually Recursive Binding and Declaration Groups

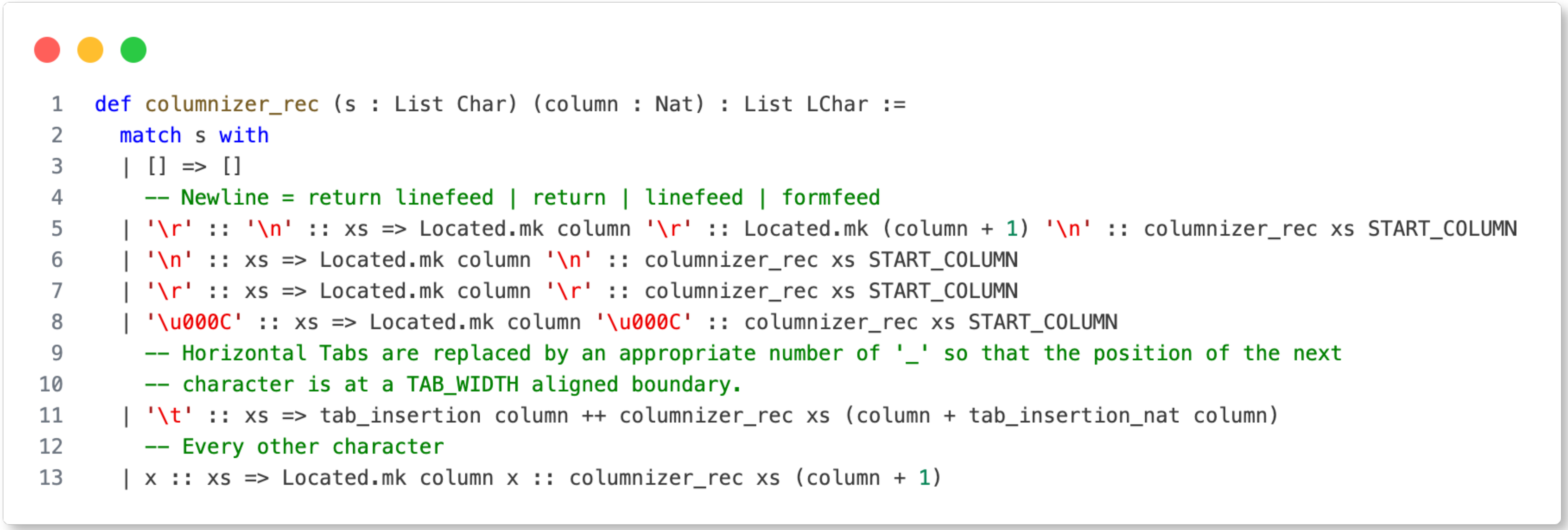
Computing Columns

The “indentation” of a lexeme is the column number of the first character of that lexeme; the indentation of a line is the indentation of its leftmost lexeme. To determine the column number, assume a fixed-width font with the following conventions:

- The characters $newline$, $return$, $linefeed$, and $formfeed$, all start a new line.
- The first column is designated column 1, not 0.
- Tab stops are 8 characters apart.
- A tab character causes the insertion of enough spaces to align the current position with the next tab stop.

For the purposes of the layout rule, Unicode characters in a source program are considered to be of the same, fixed, width as an ASCII character. However, to avoid visual confusion, programmers should avoid writing programs in which the meaning of implicit layout depends on the width of non-space characters.

Computing Columns



```
def columnizer_rec (s : List Char) (column : Nat) : List LChar :=
  match s with
  | [] => []
    -- Newline = return linefeed | return | linefeed | formfeed
  | '\r' :: '\n' :: xs => Located.mk column '\r' :: Located.mk (column + 1) '\n' :: columnizer_rec xs START_COLUMN
  | '\n' :: xs => Located.mk column '\n' :: columnizer_rec xs START_COLUMN
  | '\r' :: xs => Located.mk column '\r' :: columnizer_rec xs START_COLUMN
  | '\u000C' :: xs => Located.mk column '\u000C' :: columnizer_rec xs START_COLUMN
    -- Horizontal Tabs are replaced by an appropriate number of '_' so that the position of the next
    -- character is at a TAB_WIDTH aligned boundary.
  | '\t' :: xs => tab_insertion column ++ columnizer_rec xs (column + tab_insertion_nat column)
    -- Every other character
  | x :: xs => Located.mk column x :: columnizer_rec xs (column + 1)
```

The Lexical Syntax

- The lexical syntax of a language should ideally be defined using a regular grammar
- The lexical syntax of Haskell 2010 is almost regular
- Nested block comments are the only exception:

  ```
  {- foo {- bar -} baz -}
  ```

- Suggestion: Remove nested comments from the language report

The Lexical Syntax

Verbatim: A Verified Lexer Generator

Derek Egolf
Tufts University
Medford, MA
derek.egolf@tufts.edu

Sam Lasser
Tufts University
Medford, MA
samuel.lasser@tufts.edu

Kathleen Fisher
Tufts University
Medford, MA
kfisher@cs.tufts.edu

(FIRSTTOKEN)

$$\dfrac{\begin{array}{c} p \neq \lambda \qquad \texttt{MaxPref}_R\ p\ z \qquad p \simeq (l,e) \qquad (l,e) \in R \\ \forall r', \texttt{Index}_R\ r' < \texttt{Index}_R\ (l,e) \to \neg(p \simeq r') \end{array}}{\texttt{FirstToken}_R\ (l,p)\ z}$$

(TOKENSNIL)

$$\dfrac{\forall t, \neg\texttt{FirstToken}_R\ t\ z}{\texttt{Tokens}_R\ ([\ ], z)\ z}$$

(TOKENSCONS)

$$\dfrac{\begin{array}{c} z = p +\!\!+\, s \\ \texttt{FirstToken}_R\ (l,p)\ z \qquad \texttt{Tokens}_R\ (ts,u)\ s \end{array}}{\texttt{Tokens}_R\ ((l,p) :: ts, u)\ z}$$

Figure 4: Formal specification of the maximal munch principle applied to a string z and a list of lexical rules R. In TOKENSCONS, the unprocessed suffix is u, while in TOKENSNIL all of z is unprocessed.

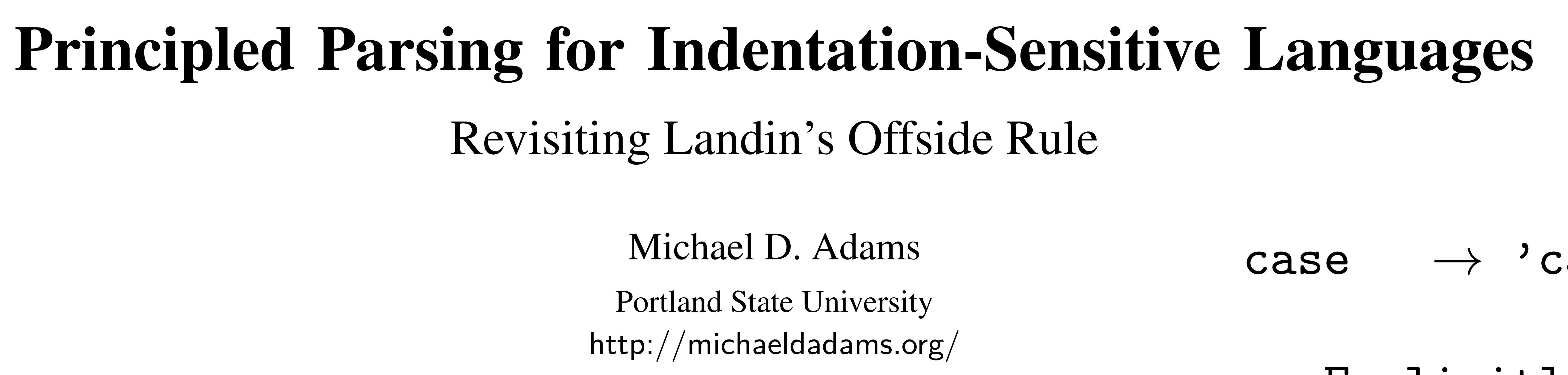
$$\begin{aligned}
\partial_a \varnothing &:= \varnothing \\
\partial_a \varepsilon &:= \varnothing \\
\partial_a [\![b]\!] &:= \texttt{if } a == b \texttt{ then } \varepsilon \texttt{ else } \varnothing \\
\partial_a (e_1 + e_2) &:= \partial_a e_1 + \partial_a e_2 \\
\partial_a (e_1 \cdot e_2) &:= (\partial_a e_1 \cdot e_2) \\
&\quad + (\texttt{if nullable } e_1 \texttt{ then } \partial_a e_2 \texttt{ else } \varnothing) \\
\partial_a (e^*) &:= \partial_a e \cdot e^*
\end{aligned}$$

Status: Completed!

Indentation-Sensitive Languages

$$
\begin{array}{lcll}
L\ (< n >: ts)\ (m : ms) & = & \mathbf{;}\ :\ (L\ ts\ (m : ms)) & \text{if } m = n \\
 & = & \}\ :\ (L\ (< n >: ts)\ ms) & \text{if } n < m \\
L\ (< n >: ts)\ ms & = & L\ ts\ ms & \\
 & & & \\
L\ (\{n\} : ts)\ (m : ms) & = & \{\ :\ (L\ ts\ (n : m : ms)) & \text{if } n > m\ (Note\ 1) \\
L\ (\{n\} : ts)\ [] & = & \{\ :\ (L\ ts\ [n]) & \text{if } n > 0\ (Note\ 1) \\
L\ (\{n\} : ts)\ ms & = & \{\ :\ \}\ :\ (L\ (< n >: ts)\ ms) & (Note\ 2) \\
 & & & \\
L\ (\}\ : ts)\ (0 : ms) & = & \}\ :\ (L\ ts\ ms) & (Note\ 3) \\
L\ (\}\ : ts)\ ms & = & \text{parse-error} & (Note\ 3) \\
 & & & \\
L\ (\{\ : ts)\ ms & = & \{\ :\ (L\ ts\ (0 : ms)) & (Note\ 4) \\
 & & & \\
L\ (t : ts)\ (m : ms) & = & \}\ :\ (L\ (t : ts)\ ms) & \text{if } m/ = 0 \text{ and parse-error}(t) \\
 & & & (Note\ 5) \\
L\ (t : ts)\ ms & = & t\ :\ (L\ ts\ ms) & \\
 & & & \\
L\ []\ [] & = & [] & \\
L\ []\ (m : ms) & = & \}\ :\ L\ []\ ms & \text{if } m \neq 0\ (Note\ 6)
\end{array}
$$

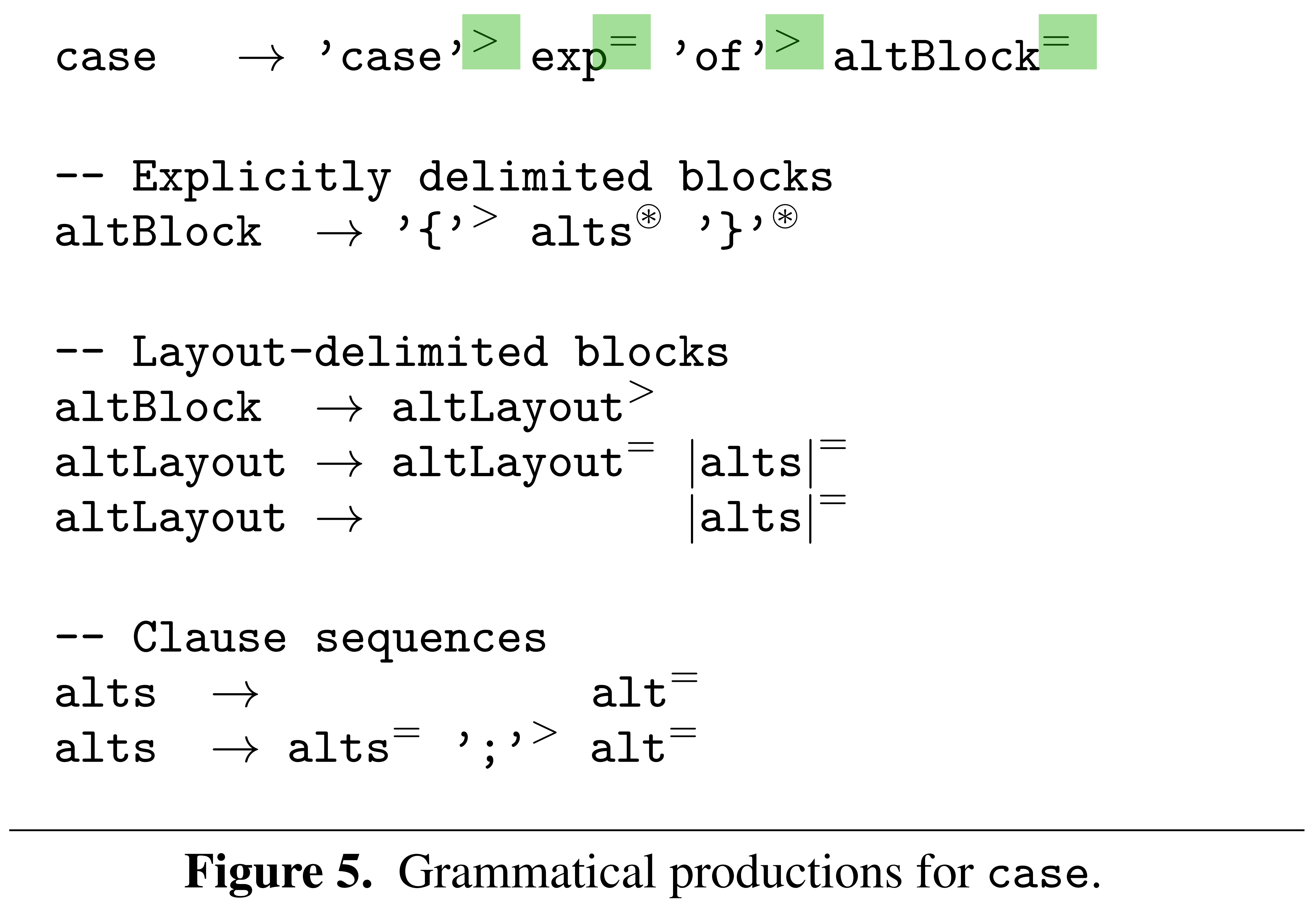
Indentation Sensitive Languages (Principled)



Principled Parsing for Indentation-Sensitive Languages

Revisiting Landin's Offside Rule

Michael D. Adams
Portland State University
http://michaeldadams.org/



$$
\begin{array}{lll}
\texttt{case} & \rightarrow & \texttt{'case'}^{>}\ \texttt{exp}^{=}\ \texttt{'of'}^{>}\ \texttt{altBlock}^{=} \\
 & & \\
\multicolumn{3}{l}{\texttt{-- Explicitly delimited blocks}} \\
\texttt{altBlock} & \rightarrow & \texttt{'\{'}^{>}\ \texttt{alts}^{\circledast}\ \texttt{'\}'}^{\circledast} \\
 & & \\
\multicolumn{3}{l}{\texttt{-- Layout-delimited blocks}} \\
\texttt{altBlock} & \rightarrow & \texttt{altLayout}^{>} \\
\texttt{altLayout} & \rightarrow & \texttt{altLayout}^{=}\ |\texttt{alts}|^{=} \\
\texttt{altLayout} & \rightarrow & |\texttt{alts}|^{=} \\
 & & \\
\multicolumn{3}{l}{\texttt{-- Clause sequences}} \\
\texttt{alts} & \rightarrow & \texttt{alt}^{=} \\
\texttt{alts} & \rightarrow & \texttt{alts}^{=}\ \texttt{';'}^{>}\ \texttt{alt}^{=}
\end{array}
$$

Figure 5. Grammatical productions for `case`.

Name Resolution and Module System

A Formal Specification of the Haskell 98 Module System

Iavor S. Diatchki, Mark P. Jones, Thomas Hallgren
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Rd, Beaverton, Oregon, USA
{diatchki,mpj,hallgren}@cse.ogi.edu

Fixity Resolution

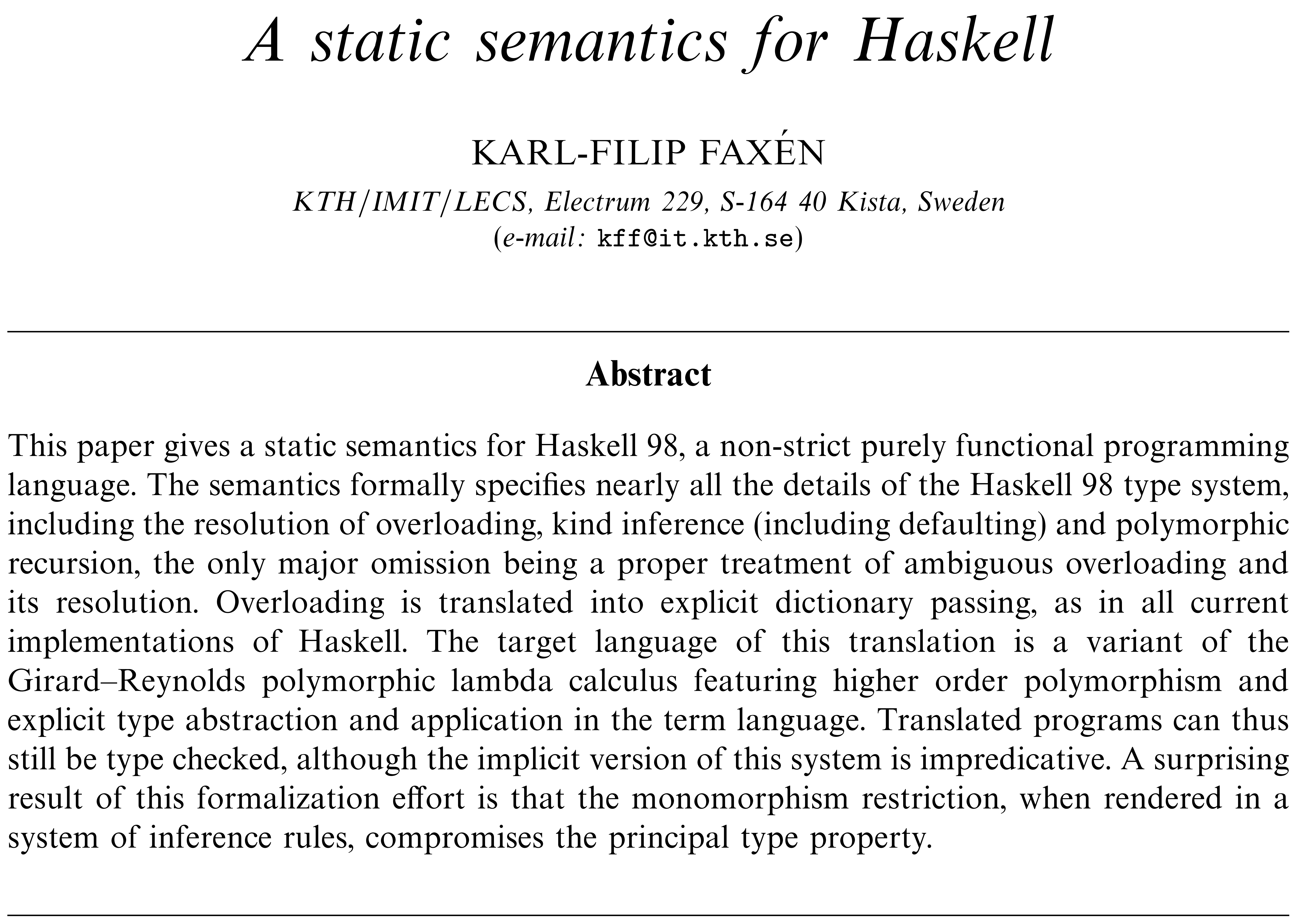
4.4.2 Fixity Declarations

gendecl	$\rightarrow$	*fixity* [*integer*] *ops*	
fixity	$\rightarrow$	`infixl` | `infixr` | `infix`	
ops	$\rightarrow$	op_1 `,` … `,` op_n	$(n \geq 1)$
op	$\rightarrow$	*varop* | *conop*	

Fixity is a property of a particular entity (constructor or variable), just like its type; fixity is not a property of that entity's *name*. For example:

Fixity can only be resolved after name resolution

Project 2:
Specifying the Type System



A static semantics for Haskell

KARL-FILIP FAXÉN

KTH/IMIT/LECS, Electrum 229, S-164 40 Kista, Sweden

(*e-mail:* `kff@it.kth.se`)

Abstract

This paper gives a static semantics for Haskell 98, a non-strict purely functional programming language. The semantics formally specifies nearly all the details of the Haskell 98 type system, including the resolution of overloading, kind inference (including defaulting) and polymorphic recursion, the only major omission being a proper treatment of ambiguous overloading and its resolution. Overloading is translated into explicit dictionary passing, as in all current implementations of Haskell. The target language of this translation is a variant of the Girard–Reynolds polymorphic lambda calculus featuring higher order polymorphism and explicit type abstraction and application in the term language. Translated programs can thus still be type checked, although the implicit version of this system is impredicative. A surprising result of this formalization effort is that the monomorphism restriction, when rendered in a system of inference rules, compromises the principal type property.

63 pages!

Declarative Type Elaboration

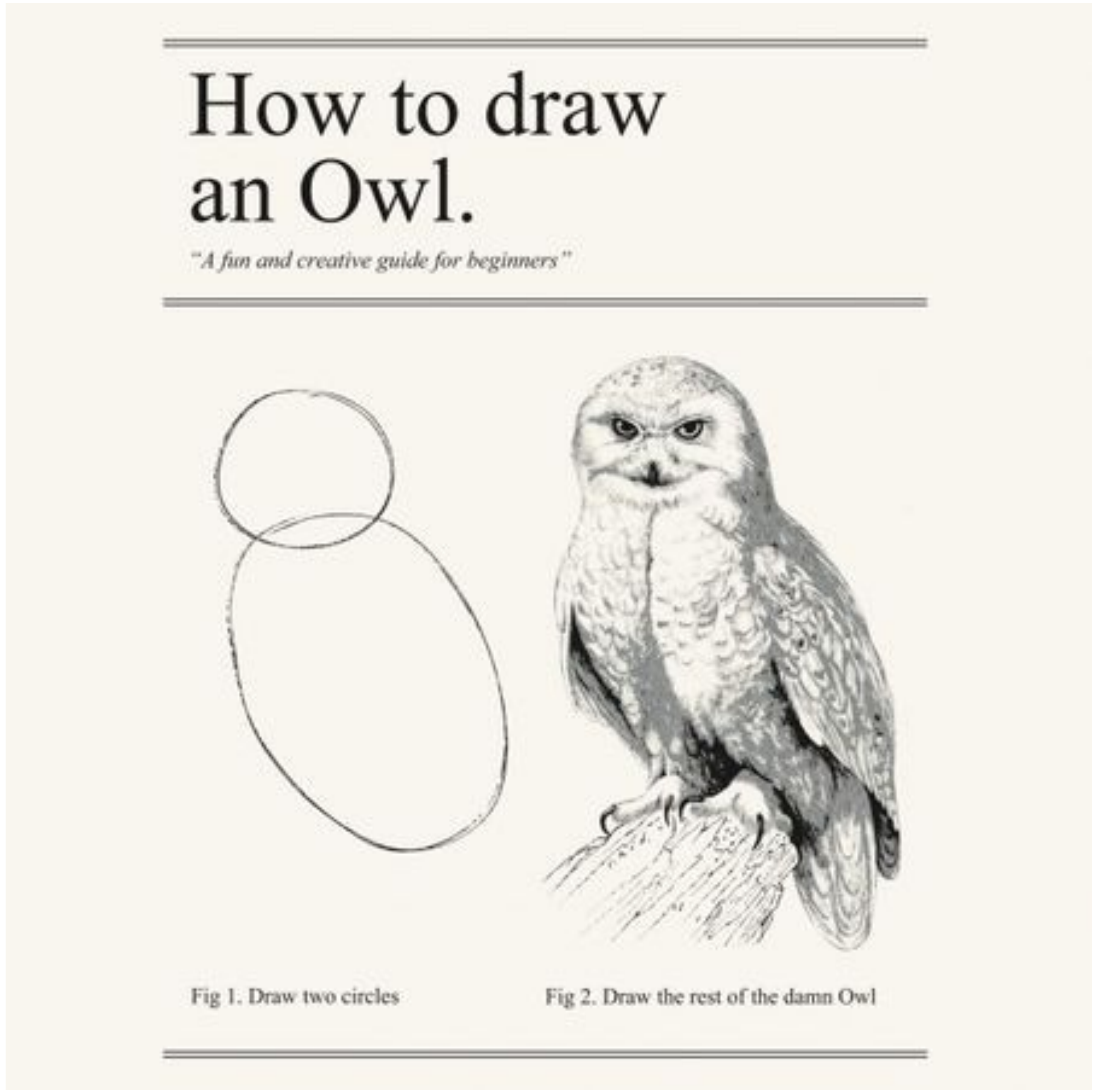
- Instead of $\Gamma \vdash t : \tau$ we use elaboration $\Gamma \vdash t \rightsquigarrow t' : \tau$
- The target language is essentially System $F\omega$
- Type classes are translated to dictionary passing style
- Example:

  ```
  f :: Show a => a -> Int
  ```

  becomes

  ```
  f :: forall a. ShowDict a -> a -> Int
  ```

How PL Theory Prepares You For the Task



How to draw an Owl.

"A fun and creative guide for beginners"

Fig 1. Draw two circles

Fig 2. Draw the rest of the damn Owl

Let Generalization

$$\frac{\overline{a} = ftv(\rho) - ftv(\Gamma) \qquad \Gamma \vdash t : \rho}{\Gamma \vdash^{poly} t : \forall \overline{a} . \rho}\ \text{GEN}$$

$$\frac{\Gamma \vdash^{poly} u : \sigma \qquad \Gamma, x : \sigma \vdash t : \rho}{\Gamma \vdash \mathtt{let}\ x\ \mathtt{=}\ u\ \mathtt{in}\ t : \rho}\ \text{LET}$$

$$\boxed{GE, IE, VE \overset{bindG}{\vdash} sigs ; bindG \rightsquigarrow \mathsf{binds} : VE_{bindG}}$$

$$\frac{\begin{array}{l}
GE, IE \oplus OE, VE \vec{\oplus} (VE_{sigs} \oplus VE_{rec}) \overset{monobinds}{\vdash} bindG \rightsquigarrow \mathsf{binds} : VE_{monobs} \\
GE \overset{sigs}{\vdash} sigs : VE_{sigs} \\
VE_{sigs} \subseteq VE_{bindG} \\
VE_{rec} = VE_{monobs} \setminus dom(VE_{sigs}) \\
\{\alpha_1, \ldots, \alpha_m\} \cap (fv(IE) \cup fv(VE)) = \emptyset \\
\mathrm{MonoRes}(bindG, dom(VE_{sigs}), \{\alpha_1, \ldots, \alpha_m\} \cap fv(OE)) \\
VE_{bindG} = \forall \alpha_1 \ldots \alpha_m . \theta \mathrel{\widetilde{\Rightarrow}} VE_{monobs} \\
OE = \mathsf{vs} \mathrel{\tilde{:}} \theta \\
VE_{monobs} = \{v_1 : \langle \mathsf{v}_1, \tau_1 \rangle, \ldots, v_n : \langle \mathsf{v}_n, \tau_n \rangle\} \\
VE_{rec} = \{v'_1 : \langle \mathsf{v}'_1, \tau'_1 \rangle, \ldots, v'_k : \langle \mathsf{v}'_k, \tau'_k \rangle\} \\
\mathsf{vs}, \mathsf{v}_{\mathsf{binds}} \text{ fresh}
\end{array}}{\begin{array}{l}
GE, IE, VE \overset{bindG}{\vdash} sigs ; bindG \\
\quad \rightsquigarrow \left\{ \begin{array}{l}
\mathsf{v}_{\mathsf{binds}} : \forall \alpha_1 \ldots \alpha_m . \widehat{\theta} \to (\tau_1, \ldots, \tau_n) \\
\qquad = \Lambda \alpha_1 \ldots \alpha_m . \lambda \mathsf{vs} \mathrel{\hat{:}} \theta . \\
\qquad\quad \mathtt{let}\ \mathsf{v}'_1 : \tau'_1 = \mathsf{v}'_1\ \alpha_1 \ldots \alpha_m\ \mathsf{vs}; \\
\qquad\qquad \ldots ; \\
\qquad\qquad \mathsf{v}'_k : \tau'_1 = \mathsf{v}'_k\ \alpha_1 \ldots \alpha_m\ \mathsf{vs} \\
\qquad\quad \mathtt{in\ let}\ \mathsf{binds} \\
\qquad\quad \mathtt{in}\ (\mathsf{v}_1, \ldots, \mathsf{v}_n); \\
\mathsf{v}_1 : \forall \alpha_1 \ldots \alpha_m . \widehat{\theta} \to \tau_1 \\
\quad = \Lambda \alpha_1 \ldots \alpha_m . \lambda \mathsf{vs} \mathrel{\hat{:}} \theta . \mathtt{case}\ \mathsf{v}_{\mathsf{binds}}\ \alpha_1 \ldots \alpha_m\ \mathsf{vs}\ \mathtt{of} \\
\qquad\qquad (\mathsf{v}_1 : \tau_1, \ldots, \mathsf{v}_n : \tau_n)\ \mathtt{->}\ \mathsf{v}_1 ; \\
\ldots ; \\
\mathsf{v}_n : \forall \alpha_1 \ldots \alpha_m . \widehat{\theta} \to \tau_n \\
\quad = \Lambda \alpha_1 \ldots \alpha_m . \lambda \mathsf{vs} \mathrel{\hat{:}} \theta . \mathtt{case}\ \mathsf{v}_{\mathsf{binds}}\ \alpha_1 \ldots \alpha_m\ \mathsf{vs}\ \mathtt{of} \\
\qquad\qquad (\mathsf{v}_1 : \tau_1, \ldots, \mathsf{v}_n : \tau_n)\ \mathtt{->}\ \mathsf{v}_n ;
\end{array} \right\} \\
\quad : VE_{bindG}
\end{array}}\ \text{BINDG}$$

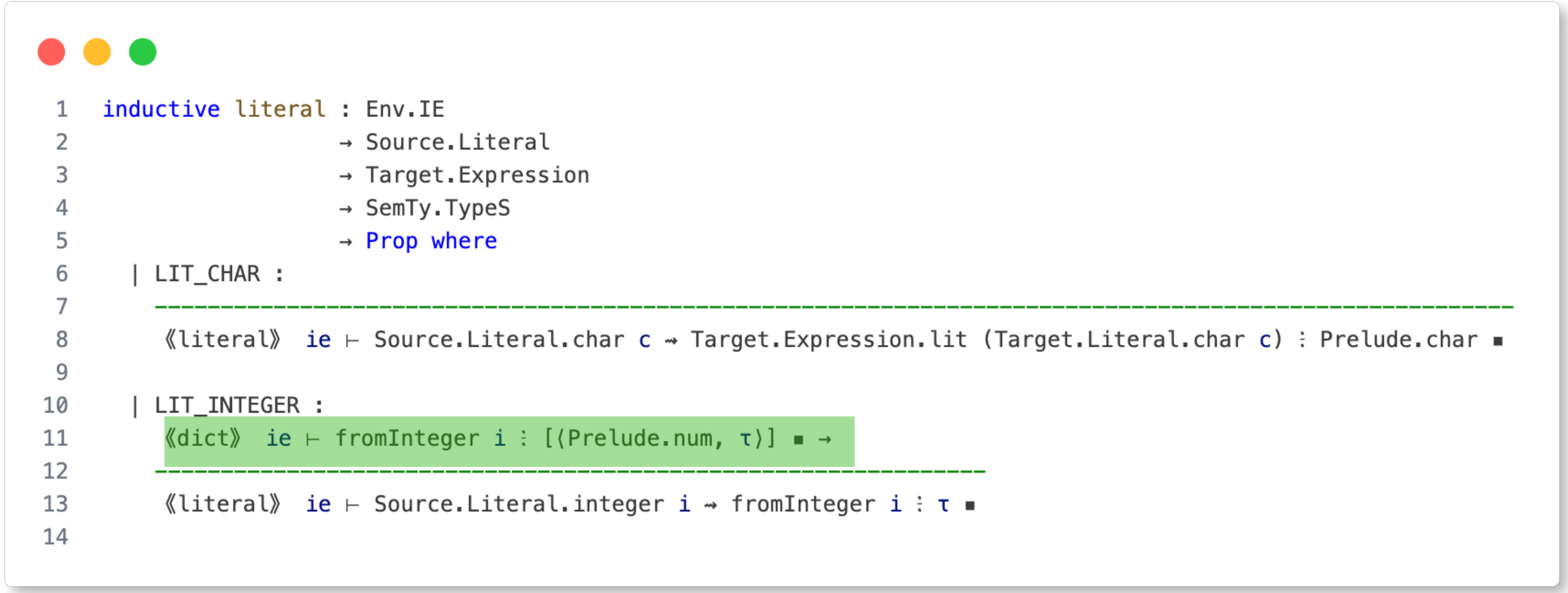
Otherwise Just A (Tedious) Translation Task

$$\boxed{IE \overset{literal}{\vdash} literal \rightsquigarrow \mathtt{e} : \tau}$$

$$IE \overset{literal}{\vdash} char \rightsquigarrow char : \mathtt{Prelude!Char}^* \qquad \text{LIT CHAR}$$

$$IE \overset{literal}{\vdash} string \rightsquigarrow string : [\mathtt{Prelude!Char}^*] \qquad \text{LIT STRING}$$

$$\frac{IE \overset{dict}{\vdash} \mathtt{e} : \mathtt{Prelude!Num}^*\ \tau}{IE \overset{literal}{\vdash} integer \rightsquigarrow \mathtt{Prelude!fromInteger}\ \tau\ \mathtt{e}\ integer : \tau} \qquad \text{LIT INTEGER}$$



```
inductive literal : Env.IE
                  → Source.Literal
                  → Target.Expression
                  → SemTy.TypeS
                  → Prop where
  | LIT_CHAR :
    ----------------------------------------------------------------------------------------------------------
    ⟪literal⟫  ie ⊢ Source.Literal.char c ⇝ Target.Expression.lit (Target.Literal.char c) ⦂ Prelude.char ▪

  | LIT_INTEGER :
    ⟪dict⟫  ie ⊢ fromInteger i ⦂ [(Prelude.num, τ)] ▪ →
    -----------------------------------------------------------------
    ⟪literal⟫  ie ⊢ Source.Literal.integer i ⇝ fromInteger i ⦂ τ ▪

```

Status of the Formalization

- Started at Zurihac 2025 and continued at Munihac 2025
- Rough guess: I am about 90% finished with the rules
- The formalization needs very careful proof reading
- Plan to present at this year's HIW (Haskell Implementor's Workshop)

Questions