

Past, Present and Future of the Haskell Language Report

Munihac, October 10th 2026

David Binder, University of Kent

If you've come for functional programming...



... or advanced mathematics ...

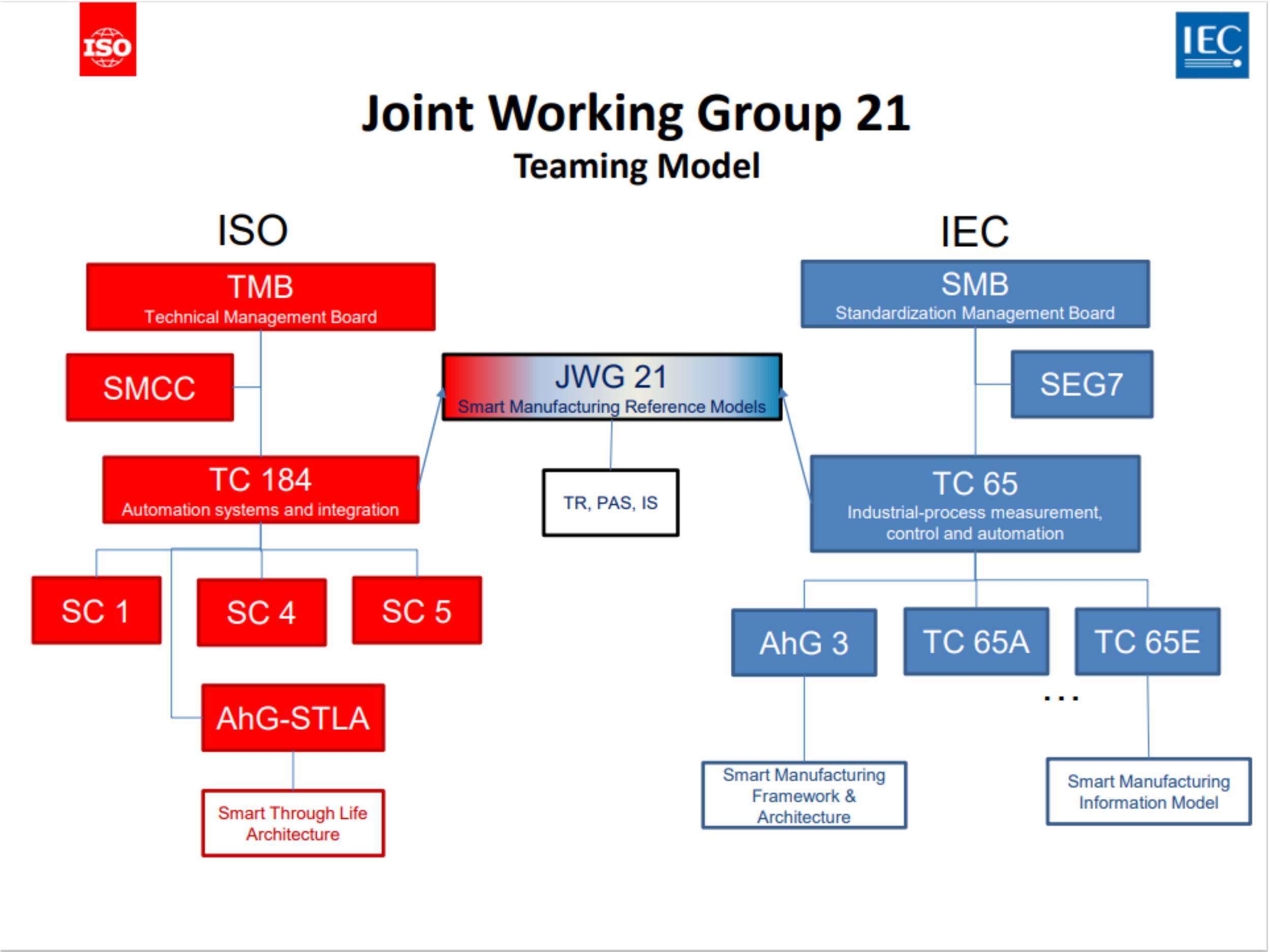
$$\begin{array}{ccccc}
 TP \otimes \pi^* T^* X & \xrightarrow{d\lambda_g \otimes (\lambda_g \times 1)} & TP \otimes \pi^* T^* X & \xrightarrow{\pi^G} & T_G P \otimes T^* X \\
 \nearrow i_J & & \nearrow i_J & & \nearrow i_C \\
 JP & \xrightarrow{j\lambda_g} & JP & & CP \\
 \downarrow \pi_J & & \downarrow \pi_J & & \downarrow \pi_{CP} \\
 \pi^*(TX \otimes T^* X) & \xrightarrow{\lambda_g \times 1} & \pi^*(TX \otimes T^* X) & \xrightarrow{\pi^G} & TX \otimes T^* X \\
 \nearrow \text{Id} & & \nearrow \text{Id} & & \nearrow \text{Id} \\
 P & \xrightarrow{\lambda_g} & P & \xrightarrow{\pi^G = \pi} & X
 \end{array}$$

Commutative diagram illustrating the relationship between various spaces and maps in advanced mathematics. The diagram shows a sequence of spaces and maps, with some maps being isomorphisms or inclusions.

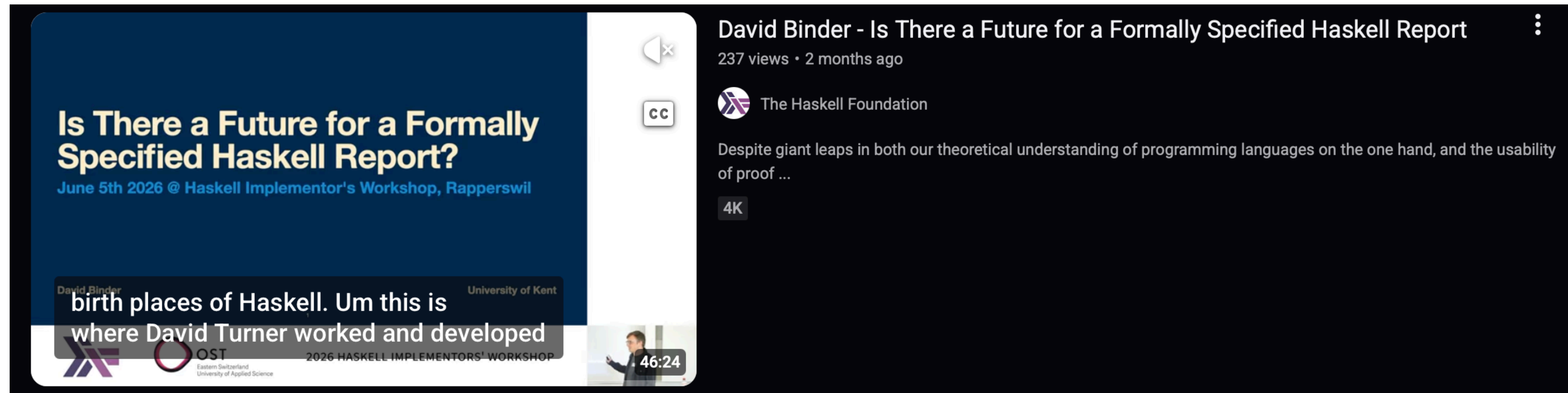
... you'll be disappointed.

STANDARDIZATION COMMITTEE WORK
gigaglitters.com

I hope you came prepared:



A Formal Haskell Report



The image shows a YouTube video player interface. The video title is "Is There a Future for a Formally Specified Haskell Report?" by David Binder. The video is 46:24 long and has 4K resolution. The description mentions "Despite giant leaps in both our theoretical understanding of programming languages on the one hand, and the usability of proof ...". The video is from The Haskell Foundation. The video player shows a thumbnail with the title "Is There a Future for a Formally Specified Haskell Report?" and the date "June 5th 2026 @ Haskell Implementor's Workshop, Rapperswil". The thumbnail also features logos for the University of Kent, OST, and the 2026 Haskell Implementors' Workshop. A subtitle on the thumbnail reads: "David Binder birth places of Haskell. Um this is where David Turner worked and developed".

Is There a Future for a Formally Specified Haskell Report?

June 5th 2026 @ Haskell Implementor's Workshop, Rapperswil

David Binder University of Kent

birth places of Haskell. Um this is where David Turner worked and developed

OST Eastern Switzerland University of Applied Science

2026 HASKELL IMPLEMENTORS' WORKSHOP

David Binder - Is There a Future for a Formally Specified Haskell Report

237 views • 2 months ago

The Haskell Foundation

Despite giant leaps in both our theoretical understanding of programming languages on the one hand, and the usability of proof ...

4K

46:24

In this talk I am not going to talk about that project, but it motivates why I want to see a new report.

Past

How did we get where we are today?

- "A History of Haskell: Being Lazy with Class" tells the story from the early beginnings to Haskell 98
- The archives of the Haskell Prime mailing list take the story up from there.

A History of Haskell: Being Lazy With Class

April 16, 2007

Paul Hudak
Yale University
paul.hudak@yale.edu

John Hughes
Chalmers University
rjmh@cs.chalmers.se

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

A Timeline of Previous Reports

- 1990: Haskell 1.0 (Eds: P. Hudak and P. Wadler, 125 pages)
- 1991: Haskell 1.1 (Eds: P. Hudak, S. Peyton-Jones, and P. Wadler, 153 pages)
- 1992: Haskell 1.2 (Eds: P. Hudak, S. Peyton-Jones, and P. Wadler, 164 pages)
- 1996: Haskell 1.3 (Eds: K. Hammond and J. Peterson, first library report)
- 1997: Haskell 1.4 (Eds: J. Peterson and K. Hammond, 139 + 73 pages)
- 1999: Haskell 98 (Eds: S. Peyton-Jones and J. Hughes, 150 + 89 pages)
- 2002: Revised Haskell 98 (Ed: S. Peyton-Jones, 260 pages, published by CUP)
- 2009: Haskell 2010 (Ed: S. Marlow, 309 pages)

Haskell 1.3 (1996)

The big one!

- Monadic IO replaces the stream-based IO that was used before.
- Type classes are generalised to higher-kinded types: Functor and Monad classes become possible.
- Newtypes, records and strictness annotations are introduced.
- Haskell looks recognisably like the language we are writing today.
- Removed one major feature which Haskell programmers won't recognise today!

In versions 1.0–1.3 of the language, every module was specified by an interface as well as an implementation. A great deal of discussion took place about the syntax and semantics of interfaces; issues such as the duplication of information between interfaces and implementations, especially when a module re-exports entities defined in one of its imports; whether one can deduce from an interface which module ultimately defines an entity; a tension between what a compiler might want in an interface and what a programmer might want to write; and so on. In the end, Haskell 1.4 completely abandoned interfaces as a formal part of the language;

A History of Haskell: Being Lazy with Class [Section 8.2]

Interface Files in Haskell 1.2

5.3 Module Interfaces

Every module has an *interface* containing all the information needed to do static checks on any importing module. All static checks on a module implementation can be done by inspecting its text and the interfaces of the modules it imports.

```
interface → interface modid where ibody

ibody      → { [iimpdecls ;] [ [fixdecls ;] itopdecls [;] ] }
             | { iimpdecls [;] }
iimpdecls  → iimpdecl1 ; ... ; iimpdecln                (n ≥ 1)
iimpdecl   → import modid ( import1 , ... , importn )
             [renaming renamings]                        (n ≥ 1)
itopdecls  → itopdecl1 ; ... ; itopdecln                (n ≥ 1)
itopdecl   → type simple = type
             | data [context =>] simple [= constrs [deriving (tycls | (tyclses))]]
             | class [context =>] class [where { icdecls [;] }]
             | instance [context =>] tycls inst
             | vars :: [context =>] type
icdecls    → icdecl1 ; ... ; icdecln                    (n ≥ 0)
icdecl     → vars :: type
```

The syntax of `interface` is similar to that of `module`, except:

- There is no export list: everything in the interface is exported.
- `import` declarations have a slightly different purpose from those in implementations (see Section 5.3.2). The list of entities to be imported is always specified explicitly.

Haskell 2010

- Hierarchical Modules
- Removal of n+k Patterns
- FFI specified in the report
- Syntax changes: Fixity resolution and Layout algorithm changed

GHC has a Haskell98 language extension, but supporting real Haskell 98 programs is likely impossible.

Haskell 2020

Announced in 2016

Dear Haskell Community!

Some time ago I asked for nominations to reboot the Haskell Prime process[1], and now I'm pleased to finally announce the formation of the new Haskell Language 2020 Committee!

The goal of the Haskell Language committee together with the Core Libraries Committee is to work towards a new Haskell 2020 Language Report. I'd like to remind everyone that the Haskell Prime Process[4] relies on *everyone* in the community to help by contributing proposals which the committee will then evaluate and if suitable help formalise for inclusion. Everyone interested in participating is also invited to join the haskell-prime mailing list.

Four years (or rather ~3.5 years) from now may seem like a long time. However, given the magnitude of the task at hand, to discuss, formalise, and implement proposed extensions (taking into account the recently enacted three-release-policy[3]) to the Haskell Report, the process shouldn't be rushed. Consequently, this may even turn out to be a tight schedule after all. However, it's not excluded there may be an interim revision of the Haskell Report before 2020.

Based on this schedule, GHC 8.8 (likely to be released early 2020) would be the first GHC release to feature Haskell 2020 compliance. Prior GHC releases may be able to provide varying degree of conformance to drafts of the upcoming Haskell 2020 Report.

Were 4 years too long rather than too short?

Haskell 2020 Fizzled Out

Haskell2020 Is Dead, but All Hope Is Not Lost

← NOVEMBER 1, 2019

HASKELL, HASKELL202X →

Haskell2020 is the long-awaited sequel to Haskell2010 — a formal, prescriptive standard of the Haskell language, that all implementations should adhere to. Today we have two previous standards, Haskell2010 and Haskell98, but neither is particularly in-line with the language as it's written in this day and age. The aim of Haskell2020 is to bring these older standards in line with the way the language is actually practiced.

reasonablypolymorphic.com/blog/haskell202x/

The good news: a lack of leadership is a very easy problem to solve. If you care about this problem, just take the reins. That's all there is to it. Anyone can do it. Yes, even you, gentle reader! Nobody needs to elevate you to a position of power. There's no admissions process. You don't need to be given any authority in order to take the lead here. This is a thing that everybody wants, but there's a coordination problem, and the only reason it hasn't been done yet is that nobody has done it.

Sandy Maguire, reasonablypolymorphic.com/blog/haskell202x/

Present

What is happening right now



[Haskell](#) [Posts](#) [Categories](#) [About](#)

Search



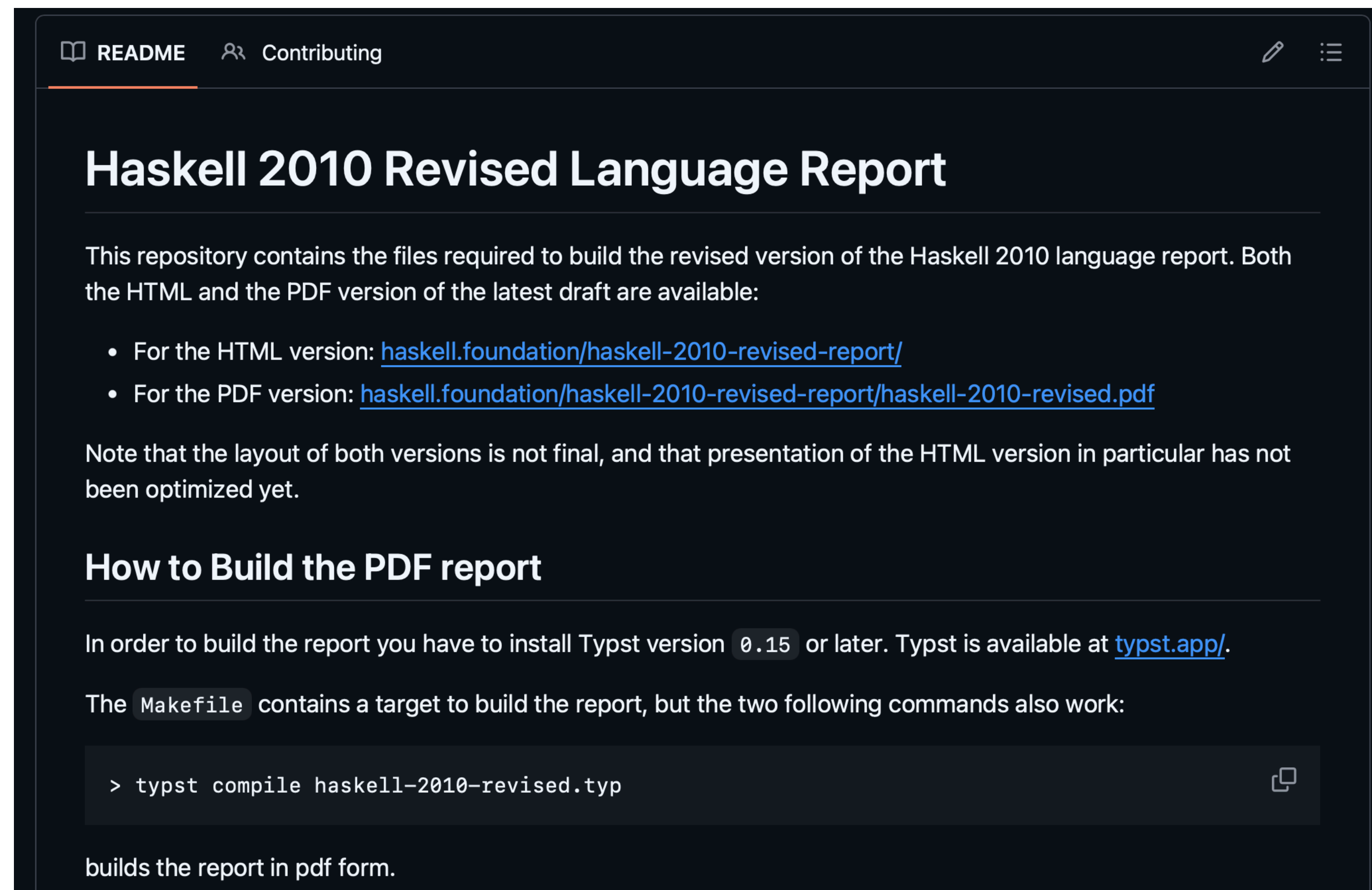
A Revised Haskell 2010 Language Report

[David Binder](#) August 04, 2026 [\[Ecosystem\]](#) [#haskell-2010](#)

The Haskell 2010 language report was released almost exactly 16 years ago, and although various attempts have been made, no further report has been released. This does not mean, however, that the language hasn't changed. On the contrary, the Haskell that we write today is quite different to the one we wrote 16 years ago, and the language has become more beautiful, consistent and ergonomic in the process. The specification, unfortunately, couldn't keep up.

Work on the Revised Report has Started

github.com/haskellfoundation/haskell-2010-revised-report



The screenshot shows the README for the Haskell 2010 Revised Language Report repository. At the top, there are tabs for 'README' and 'Contributing'. The main heading is 'Haskell 2010 Revised Language Report'. Below this, a paragraph states that the repository contains files for building the revised version of the Haskell 2010 language report, with both HTML and PDF versions available. A bulleted list provides links for the HTML and PDF versions. A note mentions that the layout is not final and the HTML version is not optimized. A section titled 'How to Build the PDF report' follows, explaining that Typst version 0.15 or later is required, with a link to typst.app/. It also mentions that the Makefile contains a target to build the report, but two commands also work. A code block shows the command 'typst compile haskell-2010-revised.typ'. Finally, it states that this command builds the report in PDF form.

README Contributing

Haskell 2010 Revised Language Report

This repository contains the files required to build the revised version of the Haskell 2010 language report. Both the HTML and the PDF version of the latest draft are available:

- For the HTML version: haskell.foundation/haskell-2010-revised-report/
- For the PDF version: haskell.foundation/haskell-2010-revised-report/haskell-2010-revised.pdf

Note that the layout of both versions is not final, and that presentation of the HTML version in particular has not been optimized yet.

How to Build the PDF report

In order to build the report you have to install Typst version `0.15` or later. Typst is available at typst.app/.

The `Makefile` contains a target to build the report, but the two following commands also work:

```
> typst compile haskell-2010-revised.typ
```

builds the report in pdf form.

The Revised Haskell 2010 WG

Name	GitHub Handle	Other
David Binder	@BinderDavid	chair
Mike Pilgrem	@mpilgrem	
Freddy Cubas	@superstar64	
Benjamin M	@L0neGamer	
Jaro Reinders	@noughtmare	
Mario Blažević	@blamario	
Mirek Kratochvil	@exaexa	co-chair
Adam Gundry	@adamgundry	
Jack Kelly	@endgame	
José Manuel Calderón Trilla	@jmct	
Brendan Lane	@gilgamec	

Goals of the Revised Haskell 2010 Report

- **Goal 1:** Remove discrepancies between the official report and what GHC does in Haskell2010 mode.
- **Goal 2:** The library report should, if at all possible, specify a subset of the base library.
- **Goal 3:** Provide the API specification of the library report in machine readable form.
- **Goal 4:** Make it easier to work on the report, remove cruft, establish processes that can be used for the next report.
- **Non-Goal:** Adopt or incorporate new language extensions.

Example 1: MonadFail

Context of the MonadFail proposal

6.3.6 The Monad Class

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a

m >> k  = m >>= \_ -> k
fail s  = error s
```

The Monad class defines the basic operations over a *monad*. See Chapter 7 for more information about monads.

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.

Finding Information about MonadFail

Control.Monad.Fail

Transitional module providing the `MonadFail` class and primitive instances.

This module can be imported for defining forward compatible `MonadFail` instances:

```
import qualified Control.Monad.Fail as Fail

instance Monad Foo where
  (>>=) = {- ...bind impl... -}

-- Provide legacy fail implementation for when
-- new-style MonadFail desugaring is not enabled.
fail = Fail.fail

instance Fail.MonadFail Foo where
  fail = {- ...fail implementation... -}
```

See <https://gitlab.haskell.org/haskell/prime/-/wikis/libraries/proposals/monad-fail> for more details.

Since: base-4.9.0.0

Copyright	(C) 2015 David Luposchainsky (C) 2015 Herbert Valerio Riedel
License	BSD-style (see the file LICENSE)
Maintainer	libraries@haskell.org
Stability	provisional
Portability	portable
Safe Haskell	Safe
Language	Haskell2010

Documentation

```
class Monad m => MonadFail (m :: Type -> Type) where
```

Source

When a value is bound in `do` -notation, the pattern on the left hand side of `<-` might not match. In this case, this class provides a function to recover.

A `Monad` without a `MonadFail` instance may only be used in conjunction with pattern that always match, such as newtypes, tuples, data types with only a single data constructor, and irrefutable patterns (`~pat`).

Instances of `MonadFail` should satisfy the following law: `fail s` should be a left zero for `>>=`,

```
fail s >>= f = fail s
```

The MonadFail proposal

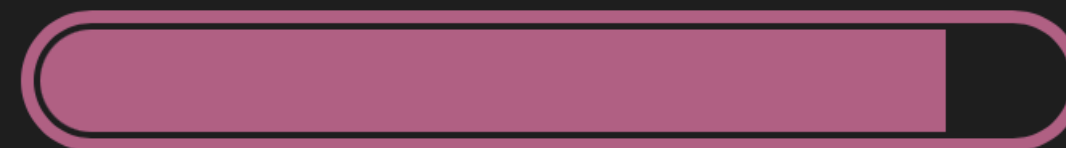
gitlab.haskell.org/haskell/prime/-/wikis/libraries/proposals/monad-fail

Making sure you're not a bot!



Powered by  Anubis

Calculating...
Difficulty: 5, Speed: 92.883kH/s



► Details

Protected by [Anubis](#) From [Techaro](#). Made with ❤️ in 🇨🇦.

Mascot design by [CELPASE](#).

This website is running Anubis version v1.26.2.

The MonadFail proposal

gitlab.haskell.org/haskell/prime/-/wikis/libraries/proposals/monad-fail

 Haskell / prime / Wiki / libraries / proposals / monad fail



monad fail



Last edited by **Herbert Valerio Riedel** 1 Aug 2018

MonadFail proposal (MFP)

A couple of years ago, we proposed to make `Applicative` a superclass of `Monad` which successfully killed the single most ugly thing in Haskell as of GHC 7.10.

Now, it's time to tackle the other major issue with `Monad` `fail` being a part of it.

For further information, feel free to ask the people responsible for or involved in this proposal:

- David Luposchainsky aka @quchen, dluposchainsky at the email service of Google
- Franz Thoma aka @fmthoma
- Herbert Valerio Riedel aka @hvr

Overview

- **The problem** - reason for the proposal
- **MonadFail class** - the solution
- **Discussion** - explaining our design choices
- **Adapting old code** - how to prepare current code to transition smoothly
- **Estimating the breakage** - how much stuff we will break
- **Transitional strategy** - how to break as little as possible while transitioning
- **Current status**

Failable Patterns

Desugaring can now be changed to produce this constraint when necessary. For this, we have to decide when a pattern match can not fail; if this is the case, we can omit inserting the `fail` call.

The most trivial examples of `unfailable patterns` are of course those that match anywhere unconditionally,

```
do x <- action    >>>    let f x = more
  more            >>>    in  action >>= f
```

Failable/Failure-Free Pattern according to GHC

```
540 {-
541 *****
542 *                                                                 *
543 * Predicates for checking things about pattern-lists in EquationInfo *
544 *                                                                 *
545 *****
546
547 \subsection[Pat-list-predicates]{Look for interesting things in patterns}
548
549 Unlike in the Wadler chapter, where patterns are either ``variables''
550 or ``constructors,'' here we distinguish between:
551 \begin{description}
552 \item[unfailable:]
553 Patterns that cannot fail to match: variables, wildcards, and lazy
554 patterns.
555
556 These are the irrefutable patterns; the two other categories
557 are refutable patterns.
558
559 \item[constructor:]
560 A non-literal constructor pattern (see next category).
561
562 \item[literal patterns:]
563 At least the numeric ones may be overloaded.
564 \end{description}
565
566 A pattern is in {\em exactly one} of the above three categories; `as'
567 patterns are treated specially, of course.
568
569 The 1.3 report defines what ``irrefutable'' and ``failure-free'' patterns are.
570 -}
```

Failable/Failure-Free Pattern according to GHC

```
629
630 -- | @isIrrefutableHsPat p@ is true if matching against @p@ cannot fail
631 -- in the sense of falling through to the next pattern.
632 --      (NB: this is not quite the same as the (silly) defn
633 --      in 3.17.2 of the Haskell 98 report.)
634 --
635 -- If isIrrefutableHsPat returns 'True', the pattern is definitely irrefutable.
636 --
637 -- However, isIrrefutableHsPat returns 'False' if it's in doubt. It's a
638 -- best effort guess with the information we have available:
639 --
640 -- - we sometimes call 'isIrrefutableHsPat' from the renamer, in which case
641 --   we don't have type information to hand. This means we can't properly
642 --   handle GADTs, nor the result TyCon of COMPLETE pragmas.
643 -- - even when calling 'isIrrefutableHsPat' in the typechecker, we don't keep
644 --   track of any long distance information like the pattern-match checker does.
645 isIrrefutableHsPat
646   :: forall p
647   .   IsPass p
648   => Bool                -- ^ Are we in a @-XStrict@ context?
649                           -- See Note [-XStrict and irrefutability]
650   -> (ConLikeP (GhcPass p) -> Bool) -- ^ How to check whether the 'ConLike' in a
651                                         -- 'ConPat' pattern is irrefutable
652   -> LPat (GhcPass p)                -- ^ The (located) pattern to check
653   -> Bool                            -- Is it irrefutable?
654 isIrrefutableHsPat is_strict irref_conLike pat = go (unLoc pat)
```

The new Report will provide a coherent definition of the MonadFail class, failable patterns, and do-notation desugaring.

Back to the Haskell 1.3 report?

A *do expression* provides a more readable syntax for monadic programming.

Translation: Do expressions satisfy these identities, which may be used as a translation into the kernel:

```
do {e} = e
do {e; stmts} = e >> do {stmts}
do {p <- e; stmts} = e >>= \p -> do {stmts}
                    where p is failure-free
do {p <- e; stmts} = let ok p = do {stmts}
                    ok _ = zero
                    in e >>= ok
                    where p is not failure-free
do {let declist; stmts} = let declist in do {stmts}
```

`>>`, `>>=`, and `zero` are operations in the classes `Monad` and `MonadZero`, as defined in the Prelude.

A failure-free pattern is one that can only be refuted by $\perp$. Failure-free patterns are defined as follows:

- All irrefutable patterns are failure-free (irrefutable patterns are described in Section 3.17.1).
- If C is the only constructor in its type, then $C\ p_1\ \dots\ p_n$ is failure-free when each of the p_i is failure free.

Example 2: Whitespace Sensitive Parsing

What the User Guide Says

16. Known bugs and infelicities

16.1. Haskell standards vs. Glasgow Haskell: language non-compliance

- Irrefutable patterns must be written in prefix form:

```
f ~a ~b = ...    -- accepted by both GHC and the Haskell Report  
f ~ a ~ b = ...  -- accepted by the Haskell Report but not GHC
```

When written in non-prefix form, `(~)` is treated by GHC as a regular infix operator.

See [GHC Proposal #229](#) for the precise rules.

Looks like an innocent change!

GHC Proposal #229

Whitespace-sensitive operator parsing (~, !, @, , \$, -)

```
.. date-accepted:: 2019-08-24
```

```
.. author:: Vladislav Zavialov
```

```
.. ticket-url:: https://gitlab.haskell.org/ghc/ghc/-/merge\_requests/1664
```

```
.. implemented:: 9.0, with the warnings later added in 9.2
```

GHC Proposal #229

Proposed Change Specification

- Classify lexical non-terminals (lexemes and whitespace) of the Haskell Report into three groups:
 - Opening non-terminals include identifiers (`qvarid`, `qconid`, `qvarsym`, `qconsym`), keywords (`if`, `do`, `case`, `let`, and so on), literals `literal`, and opening brackets `(`, `[`, `{`.
 - Closing non-terminals include identifiers, keywords, literals, and closing brackets `)`, `]`, `}`.
 - Other non-terminals are any lexical non-terminals not considered opening or closing, in particular `whitespace` (including comments), and separators `,`, `;`.

Note that identifiers, keywords, and literals are classified as both closing and opening.

Language report currently doesn't distinguish between lexemes and non-terminals.

GHC Proposal #229

- Any unqualified operator (`varsym` or `consym`) is interpreted as "prefix", "suffix", "tight infix", or "loose infix", based on the preceding and following lexical non-terminals:
 - Prefix occurrence: not(closing), operator, opening
 - Suffix occurrence: closing, operator, not(opening)
 - Tight infix occurrence: closing, operator, opening
 - Loose infix occurrence: not(closing), operator, not(opening)

The general principle can be demonstrated as follows:

```
a ! b    -- a loose infix occurrence
a!b      -- a tight infix occurrence
a !b     -- a prefix occurrence
a! b     -- a suffix occurrence
```



We need to specify an additional preprocessing pass to classify operators before parsing.

**The new report will properly define
whitespace sensitive operator parsing
that can be implemented by other tools.**

What is the Timeline for the New Report?

- Working Group with 11 members has been established on August 21st
- Several changes to the report have been merged: "forall" is now a keyword, syntax of numeric literals, recursive modules, removal of data types that use datatype contexts from the library report.
- We are working in parallel on many of the changes to the report.
- We will hopefully announce a beta version of the new report before the new year.

Future

**What next after the revised
report?**

What value can only the language report provide?

- Develop and document coherent language principles instead of individual features.
- Trickle-down documentation: Language Specification -> Books and Tutorials
- "Real World Haskell (Haskell 2026 edition)"
- We can develop against a specification instead of an implementation.

How to Guarantee Success of the Next Report

- Define a clear scope at the beginning instead of discovering it in the process.
- Set a clear timeline of 6 months to 1 year to keep the end in sight.
- Standardization committee is wrong place for novel features or research
- Work on the textual source of the report instead of independent proposal documents.
- Aim for delimited increments instead of monolithic integration.
- Many different well-scoped projects are possible: Not just the specification of GHC2021 or GHC2024.

Proposal 1: Specify Concurrent Haskell and the Memory Model

Concurrent Haskell and the Memory Model

- The papers which introduce Concurrent Haskell and STM are not up-to-date with what is implemented.
- The exception model of the language report is too naive: The report only knows $\perp$.
- Haskell does not have a memory model which specifies the interaction of mutable references (IORefs) and concurrency (forkIO)
- "Relaxed memory models must be rigorous", Nardelli et al: C, Java, OCaml, ... all have formally verified and machine checked memory models.

Sequential Consistency?



SC-Haskell: Sequential Consistency in Languages That Minimize Mutable Shared Heap

Michael Vollmer¹ Ryan G. Scott¹ Madanlal Musuvathi² Ryan R. Newton¹

Indiana University (USA)¹, Microsoft Research (USA)²

{vollmer, rgscott, rrnewton}@indiana.edu madan@microsoft.com

Haskell could be the first mainstream language with a SC memory model.

Proposal 2: New Baseline for Syntax

Baseline for Syntax

- What is the difference between "NegativeLiterals" and "LexicalNegation"
- What are the rules for prefix and postfix operators, sections, commata in enumerations.
- What is the syntax and meaning of numeric and string literals? Currently governed by ~10 different extensions.

Proposal 3: Visible Modern Kinds

Visible Modern Kinds

- The Haskell 2010 report only knows kinds $*$ and $\kappa \rightarrow \kappa$
- Kinds are not part of the visible syntax in Haskell 2010: Kinds cannot be annotated
- Since there are no kind variables Haskell 2010 has to resort to *kind defaulting* instead of generalization (Cp. PolyKinds)
- The Haskell 2010 kind system doesn't know about unlifted or unboxed types.
- A specification of GHC's modern *kind system* is much simpler than the specification of GHC's *type system*.

Proposal 4: Standardize Mutually-Recursive and Parameterized Modules

Module Interfaces

- Haskell 2010 requires conformant compilers to implement mutually recursive modules, but doesn't specify how to do so.
- GHC uses the .hs-boot files to allow for recursion in modules
- Backpack uses signature files to specify module signatures.
- The syntax of these files is very similar.

5.3 Module Interfaces

Every module has an *interface* containing all the information needed to do static checks on any importing module. All static checks on a module implementation can be done by inspecting its text and the interfaces of the modules it imports.

```
interface → interface modid where ibody

ibody      → { [iimpdecls ;] [ [fixdecls ;] itopdecls [;] ] }
            | { iimpdecls [;] }
iimpdecls  → iimpdecl1 ; ... ; iimpdecln                (n ≥ 1)
iimpdecl   → import modid ( import1 , ... , importn )
            [renaming renamings]                        (n ≥ 1)
itopdecls  → itopdecl1 ; ... ; itopdecln                (n ≥ 1)
itopdecl   → type simple = type
            | data [context =>] simple [= constrs [deriving (tycls | (tyclses))]]
            | class [context =>] class [where { icdecls [;] }]
            | instance [context =>] tycls inst
            | vars :: [context =>] type
icdecls    → icdecl1 ; ... ; icdecln                    (n ≥ 0)
icdecl     → vars :: type
```

The syntax of *interface* is similar to that of *module*, except:

- There is no export list: everything in the interface is exported.
- import declarations have a slightly different purpose from those in implementations (see Section 5.3.2). The list of entities to be imported is always specified explicitly.

**Will specification become more
important again?**

Your Q's my A's

Your ideas for a new report