HASKELL – BEING LAZY 101

TÓPICOS ABORDADOS NESTA APRESENTAÇÃO

- História
- Por que usar Haskell?
- Onde Haskell é usado?
- Pontos negativos*

HISTÓRIA

BACKGROUND DO PARADIGMA FUNCIONAL

(Década de 50) John McCarthy inventou Lisp

(Década de 60) Peter Landin e Christopher Strachey identificaram a importância do calculo lambda para modelar linguagens de programação.

(Década de 70) Rod Burstall e John Darlington estavam fazendo transformação de programas em linguagem funcional de primeira ordem usando definição de funções por *pattern matching*

(1976) David Turner desenvolveu uma linguagem de alto nível que incorporou as ideias de Burstall e Darllington sobre *pattern* matching em uma linguagem de programação executável (SASL)

(Década de 70) Gerry Sussman e Guy Steele desenvolveram Scheme tratando melhor o cálculo lambda no dialeto Lisp. Robin Milner inventou ML.

(1978) John Backus - “Can programming be liberated from the von Neumann style?”

HISTÓRIA

BACKGROUND DA AVALIAÇÃO PREGUIÇOSA

Uma série de publicações entre as décadas de 70 e 80 alavancaram o interesse na ideia de linguagens funcionais que usam avaliação preguiçosa (*lazy* ou *non-strict* ou call-by-need) como veículo para escrever programas sérios.

(1976) David Wise publicou "Cons should not evaluate its arguments" que levou a avaliação preguiçosa para Lisp.

(1976) Peter Henderson e James H. Morris Jr publicaram "A lazy evaluator" onde eles citaram Vuillemin e Wadsworth como responsáveis pela ideia, popularizaram ela no POPL (*Principles of Programming Languages*) e atribuiram o nome ao conceito.

(Década de 80) David Turner, utilizando as linguagens SASL e KRC, mostrou a elegancia da programação que usa avaliação preguiçosa. Em particular o uso de listas preguiçosas para emular diversos comportamentos.

(Década de 80) Em software, uma série de técnicas baseadas em redução a grafos estavam sendo exploradas

HISTÓRIA

BACKGROUND DA AVALIAÇÃO PREGUIÇOSA

(Década de 80) A possibilidade de que isso pudesse levar a arquiteturas que não seguiam o padrão von Neumann potencializou os holofotes sobre esse conceito

(1981) Um curso de nome *Advanced Course on Functional Programming and its Applications* foi ministrado por Peter Henderson, John Darlington, and David Turner no qual muitos nomes importantes da cena da computação estavam presentes e foi considerado um momento divisor de águas para a maior parte deles.

(1981) A primeira conferencia chamada *Functional Programming Languages and Computer Architecture (FPCA)* aconteceu Portsmouth, New Hampshire. Essa conferencia se tornou uma conferencia chave na área.

(Década de 80) Duas conferencias de Lisp ocorrem. Nelas algumas importantes apresentaçãoes são feitas a em favor da programação funcional. A conferencia de Lisp foi posteriormente fundida à FPCA e se tornam a *International Conference on Functional Programming (*ICFP) em 1996.

(1987) Ham Richads e Dave Turner aulas com o tema Progamação Declarativa para o “Ano da Programação” organizado pela *University of Texas*. A maior parte tratava de Programação Funcional Preguiçosa com aulas pratricas usando Miranda. Esse encontro, demonstrando a simplicidade e elegancia da programação funcional, cativou muitos autores e pesquisadores.

HISTÓRIA

BACKGROUND DA AVALIAÇÃO PREGUIÇOSA

Conexão direta com calculo lambda puro utilizando passagem por referencia

Avaliação preguiçosa

Possibilidade de Manipular e Representar estruturas de dados infinitas

Tecnicas de implementação viciantes por serem simples e bonitas

HISTÓRIA

A TORRE DE BABEL

Miranda

- David Turner
- Tipagem Polimórfica Forte
- Inferencia de Tipos

Orwell

- Wadler
- Wadler e Bird escreveram um livro que tentou evitar esse efeito de Torre de Babel

Lazy ML (LML)

- Augustsson, Johnsson e mais tarde Peyton Jones
- Programas funcionais preguiçosos podem ser compilados favorecendo a eficiencia do Código

Alfl

- Hudak

Ponder

- Jon Fairbairn

Clean

- Rinus Plasmeijer e colegas
- Linguagem baseada explicitamente em redução a grafos

Id

- Arvind e Nikhil
- Linguagem não estrita de fluxo de dados

Daisy

- Cordelia Hall, John O'Donnell, e colegas
- Dialeto preguiçoso de Lisp

HISTÓRIA
A TORRE DE BABEL

- Em geral, todas essas linguagens estavam sendo desenvolvidas isoladamente
- **Individualmente lhes faltavam:**
  - Esforço em seu design
  - Implementações diversificadas
  - Usuários
- Pouquíssimas razões para destacar superioridade de alguma delas em relação às outras
  - Tirando a sintaxe, elas eram bem parecidas

HISTÓRIA

A ORIGEM

- Comunicação mais rápida de idéias
- Fundação estável para desenvolvimento de aplicações reais
- Encorajar o uso do paradigma funcional

- Linguagem Comum a todas essas outras que estavam sendo desenvolvidas
- Não Estrita
- Preguiçosa
- Puramente Funcional
- Liberdade para extender e modificar a linguagem
- Liberdade para construir e distribuir uma implementação

HISTÓRIA

A ORIGEM

The Yale Meeting The first physical meeting (after the impromptu FPCA meeting) was held at Yale, January 9–12, 1988, where Hudak was an Associate Professor. The first order of business was to establish the following goals for the language:

1. *It should be suitable for teaching, research, and applications, including building large systems.*
2. *It should be completely described via the publication of a formal syntax and semantics.*
3. *It should be freely available.* Anyone should be permitted to implement the language and distribute it to whomever they please.
4. *It should be usable as a basis for further language research.*
5. *It should be based on ideas that enjoy a wide consensus.*
6. *It should reduce unnecessary diversity in functional programming languages.* More specifically, we initially agreed to base it on an existing language, namely OL.

Algumas dessas ideias foram abandonadas

- Basear explicitamente a linguagem em OL
- Só adicionar idéias bem embasadas quando adicionaram type classes
- Uma semantica formal não foi desenvolvida

- Um nome precisava ser escolhido

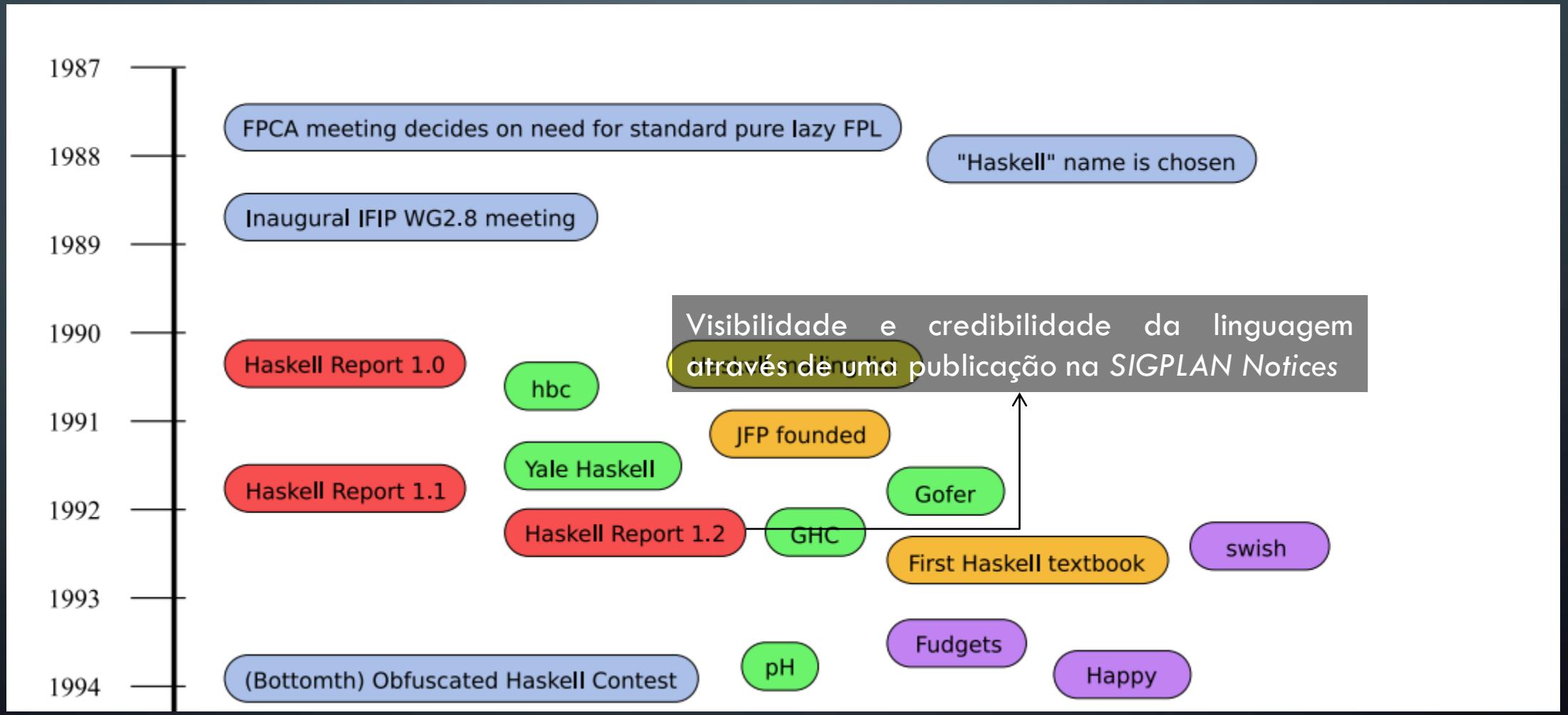
Estava criado o Haskell Comitee

HISTÓRIA

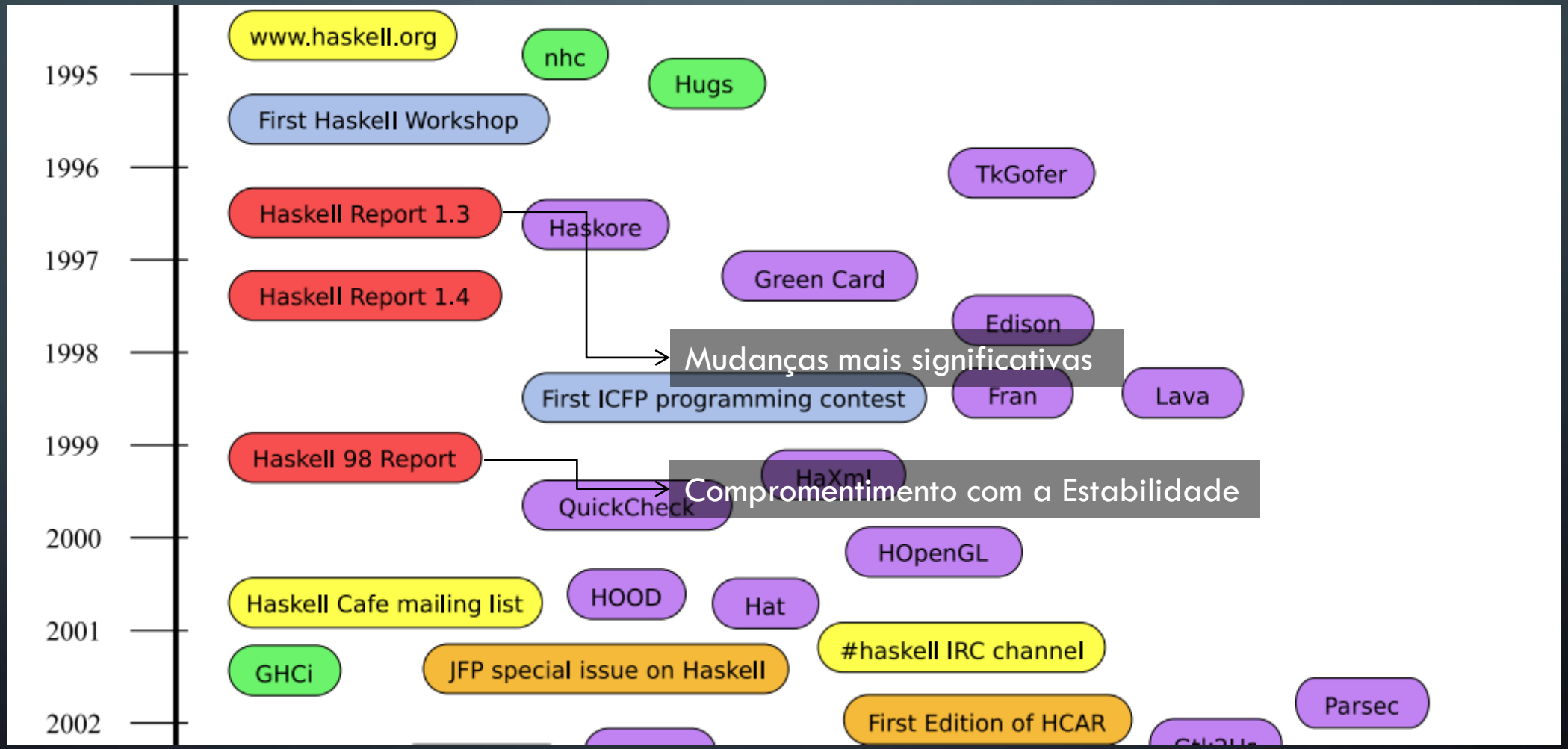
A ORIGEM

- Alguns outros encontros foram realizados antes do lançamento do primeiro artigo sobre haskell

September 1987. Initial meeting at FPCA, Portland, Oregon.
December 1987. Subgroup meeting at University College London.
January 1988. A multi-day meeting at Yale University.
April 1988. A multi-day meeting at the University of Glasgow.
July 1988. The first IFIP WG2.8 meeting, in Glasgow.
May 1989. The second IFIP WG2.8 meeting, in Mystic, CT.



Visibilidade e credibilidade da linguagem através de uma publicação na *SIGPLAN Notices*


1995
1996
1997
1998
1999
2000
2001
2002
www.haskell.org
nhc
Hugs
First Haskell Workshop
TkGofer
Haskell Report 1.3
Haskore
Green Card
Haskell Report 1.4
Edison
Mudanças mais significativas
First ICFP programming contest
Fran
Lava
Haskell 98 Report
HaXml
Comprometimento com a Estabilidade
QuickCheck
HOpenGL
Haskell Cafe mailing list
HOOD
Hat
#haskell IRC channel
GHCi
JFP special issue on Haskell
Parsec
First Edition of HCAR

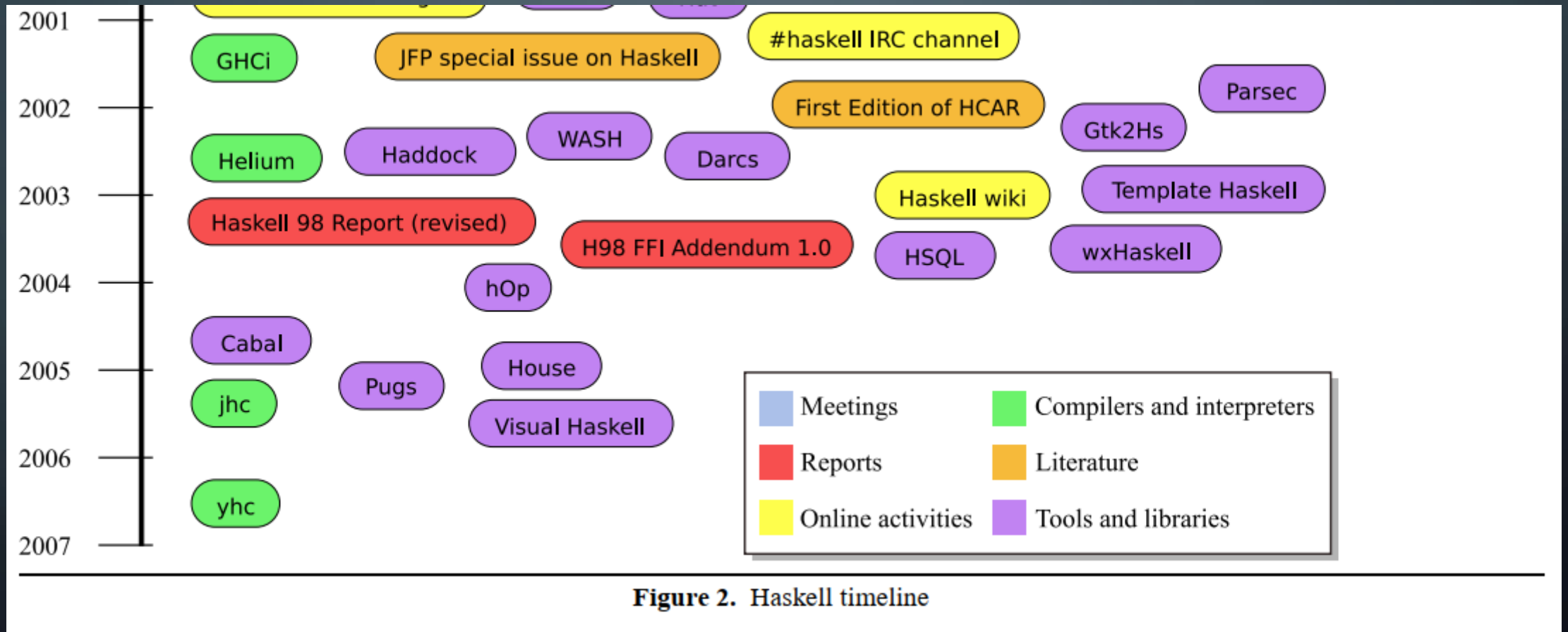


Figure 2. Haskell timeline

POR QUE USAR HASKELL?

- Pureza:

Diferente de outras linguagens de programação funcional, Haskell é puro. Assim, funções não podem ter nenhum efeito colateral. Com isso, o comportamento de uma função será sempre o mesmo para um dado input. Uma outra consequência desta característica é que funções podem ser analisadas em paralelo com segurança.

- Tipagem forte

- Haskell fornece uma maneira elegante, concisa e segura de escrever seus programas. Programas não irão travar inesperadamente, nem retornar uma saída estranha.
- Haskell tem tipagem forte. Não é possível converter um Double para um Inteiro acidentalmente, ou seguir um null pointer. Fazer conversões manualmente entre tipos isomórficos pode ser trabalhoso, mas ainda pode ser uma medida boa de segurança.
- Linguagens que fazem tais conversões automaticamente podem gerar problemas sérios.

- Além disso, tipos podem ser inferidos automaticamente. Ao analisar como as variáveis são utilizadas, Haskell irá deduzir qual o tipo mais apropriado para a variável, e a partir disso o type-checking será feito, para garantir que não haja incompatibilidade com outros pedaços de código.

- Elegância

- Mesmo que isso não signifique muito em termos de estabilidade ou desempenho, a elegância de Haskell é um ponto positivo. Para simplificar: as coisas funcionam como você esperaria que elas funcionassem.
- A seguir, vamos comparar códigos de quicksort em Haskell e C++:

```
template <typename T>
void qsort (T *result, T *list, int n)
{
    if (n == 0) return;
    T *smallerList, *largerList;
    smallerList = new T[n];
    largerList = new T[n];
    T pivot = list[0];
    int numSmaller=0, numLarger=0;
    for (int i = 1; i < n; i++)
        if (list[i] < pivot)
            smallerList[numSmaller++] = list[i];
        else
            largerList[numLarger++] = list[i];

    qsort(smallerList,smallerList,numSmaller);
    qsort(largerList,largerList,numLarger);

    int pos = 0;
    for ( int i = 0; i < numSmaller; i++)
        result[pos++] = smallerList[i];

    result[pos++] = pivot;

    for ( int i = 0; i < numLarger; i++)
        result[pos++] = largerList[i];

    delete [] smallerList;
    delete [] largerList;
```

```
qsort :: (Ord a) => [a] -> [a]
 qsort []     = []
 qsort (x:xs) = qsort less ++ [x] ++ qsort more
     where less = filter (<x)  xs
           more = filter (>=x) xs
```

- Haskell: Menos bugs

- Além dos tópicos mostrados acima, programas feitos Haskell tem menos problemas pelos seguintes motivos:

- **Gerenciamento de memória -** O programador só precisa se preocupar com a implementação do algoritmo, pois o GC cuida do gerenciamento da memória.

- **Modularidade -** Frequentemente, funções podem ser comprovadas como corretas por indução. Ao combinar duas funções corretas, temos um resultado correto (supondo que a combinação esteja correta).

QUANDO C É MELHOR?

Na vida, nem tudo são flores. Existem áreas em que Haskell não é adequado.

- Em aplicações em que o desempenho é necessário a qualquer custo, ou quando o objetivo é um ajuste detalhado de um algoritmo de baixo nível, uma linguagem imperativa como C provavelmente seria uma escolha melhor do que Haskell, exatamente porque fornece um controle mais íntimo sobre a maneira como a computação é realizada.
- Assim, enquanto não aparece um compilador suficientemente inteligente que seja capaz de derivar o equivalente C do one-liner Haskell por si só, uma outra linguagem imperativa é mais indicada.

PONTOS NEGATIVOS (DE ACORDO COM A COMUNIDADE)

Algumas reclamações feitas pelos usúarios do r/Haskell (que, atualmente tem 34.700 usúarios, com uma media de 550 usúarios online simultaneamente):

- A compilação geralmente leva muito tempo.
- A qualidade e a cobertura da biblioteca são inconsistentes.
  - Tipos não substituem documentação. Muitas bibliotecas não possuem documentação adequada, com o pretexto de que “tipos já são suficientes para dizer o que uma função deve fazer).
  - Existem muitas funções da biblioteca PRELUDE (O Prelude é importado por padrão em todos os módulos Haskell, a menos que haja uma declaração de importação explícita para ele, ou a extensão NoImplicitPrelude esteja ativada) que não são totais.

PONTOS NEGATIVOS (DE ACORDO COM A COMUNIDADE)

```
λ head []
*** Exception: Prelude.head: empty list
```

- String, Bytestring e Text – Existem diferentes tipos para representar texto. Bibliotecas diferentes usam tipos diferentes. Isso significa que um programa precisa ter muito código apenas para cuidar da conversão entre os tipos de texto.
- Os campos de um record não podem ser reutilizados: por exemplo, se dois tipos registro diferentes quisessem usar o mesmo campo de registro "hitPoints :: int", haveria um conflito.

- **Avaliar a complexidade** - Em grande parte devido à avaliação preguiçosa do Haskell, medir a complexidade de espaço e tempo rapidamente é totalmente diferente do que você esperaria se estivesse vindo de uma linguagem avidamente avaliada.
- **Não existem tipos dependentes em Haskell** - Haskell continua evoluindo para tipos mais expressivos, mas o coração da linguagem não é projetado para isso. Um dos motivos, de acordo com o wiki de haskell, é que existem algumas desvantagens em adotar tipos dependentes, como a indecidibilidade da inferência de tipos para tipos dependentes.

ONDE HASKELL É USADO?

- Na Indústria:

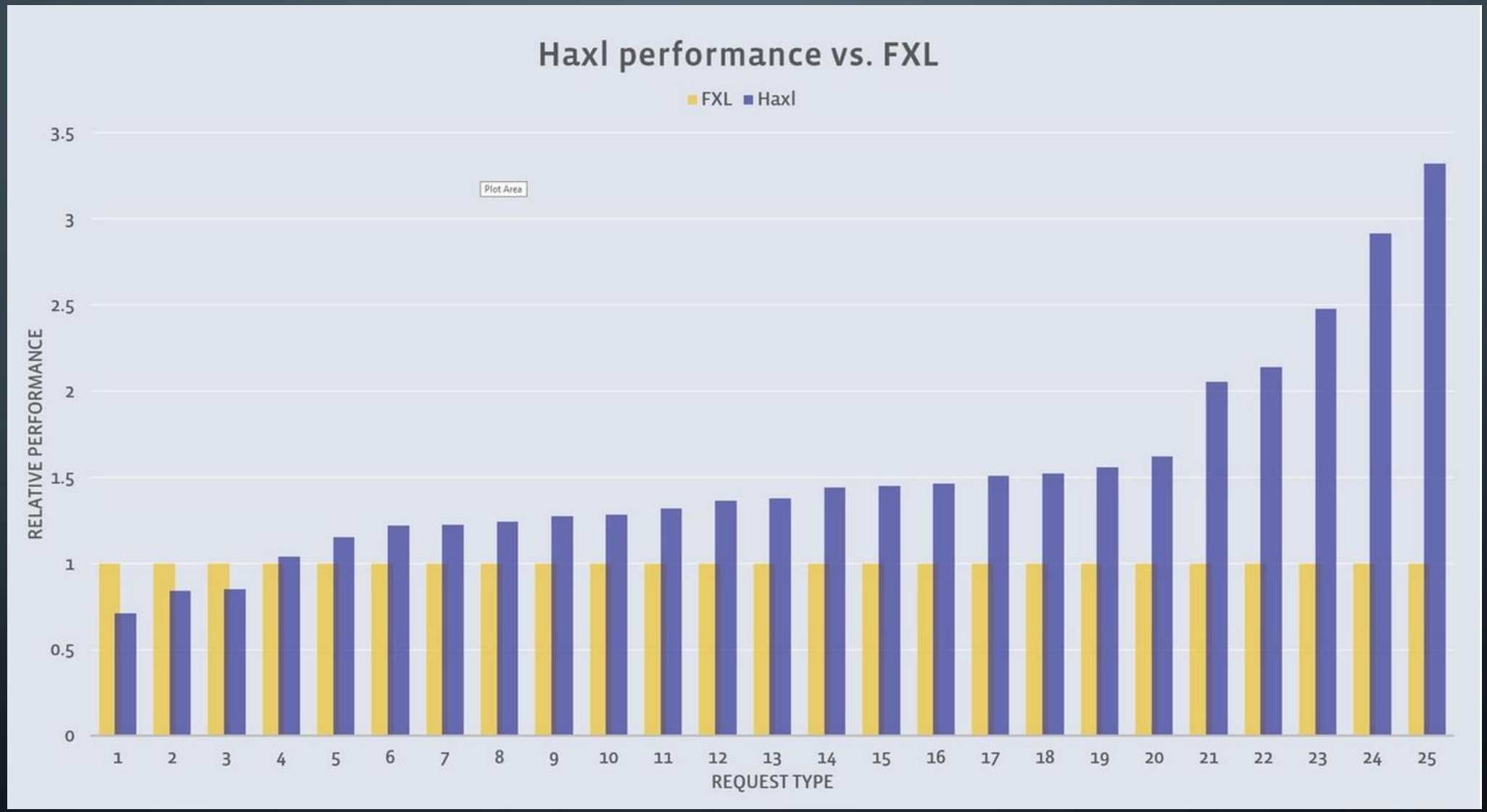
- AT&T - Haskell está sendo usado na divisão de segurança de rede para automatizar o processamento de reclamações de abuso de internet. Haskell os permitiu cumprir facilmente prazos apertados com resultados confiáveis.
- ABN AMRO Amsterdam, Holanda - O ABN AMRO é um banco internacional sediado em Amsterdã. Para suas atividades de banco de investimento, ele precisa mensurar o risco de contraparte em carteiras de derivativos financeiros.
- Detexify – É um sistema de reconhecimento de escrita on-line, que ajuda você a obter a codificação de símbolos no LaTeX. Seu backend está escrito em Haskell.

- Aplicativos:

- Elm Compiler – O compilador de Elm (outra linguagem puramente funcional) é escrito em Haskell.
- Pandoc – Conversor universal de documentos.
- Xmonad - é um gerenciador de janelas X11 dinâmico para Linux.
- QuickCheck - Originalmente escrito em Haskell, é usado para gerar casos de teste para suítes de teste.

- Outros:
  - Elm Compiler – O compilador de Elm (outra linguagem puramente funcional) é escrito em Haskell.
  - Pandoc – Conversor universal de documentos.
  - Xmonad - é um gerenciador de janelas X11 dinâmico para Linux.
  - QuickCheck - Originalmente escrito em Haskell, é usado para gerar casos de teste para suítes de teste.
  - Haskabelle – É um conversor de arquivos Haskell para Isabelle/teorias HOL implementadas em Haskell.
  - Paradox - Processa problemas lógicos de primeira ordem e tenta encontrar modelos de domínio finito para eles.
  - FOL Resolution Theorem Prover - Uma implementação de um simples provador de teoremas em lógica de primeira ordem usando Haskell.

- Através de um rule engine (que roda um conjunto de regras chamado de políticas) chamado de Sigma, Facebook tem combatido spam em suas redes.
- a cada interação feita, Sigma analisa um conjunto de políticas específicas, ajudando na identificação e bloqueio de atividades maliciosas.
- Previamente escrito em **FXL**, linguagem desenvolvida com essa finalidade, Sigma não era ideal para expressar as políticas complexas que cresciam rapidamente. Com isso em mente, desenvolvedores decidiram trocar para Haskell, e decidiram montar um framework encima dele para complementar o que ainda precisavam. Isto ficou conhecido como **Haxl**.


Haxl performance vs. FXL
FXL
Haxl
Plot Area
RELATIVE PERFORMANCE
3.5
3
2.5
2
1.5
1
0.5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
REQUEST TYPE