Compiling functional code for system-level environment

Miroslav Kratochvíl

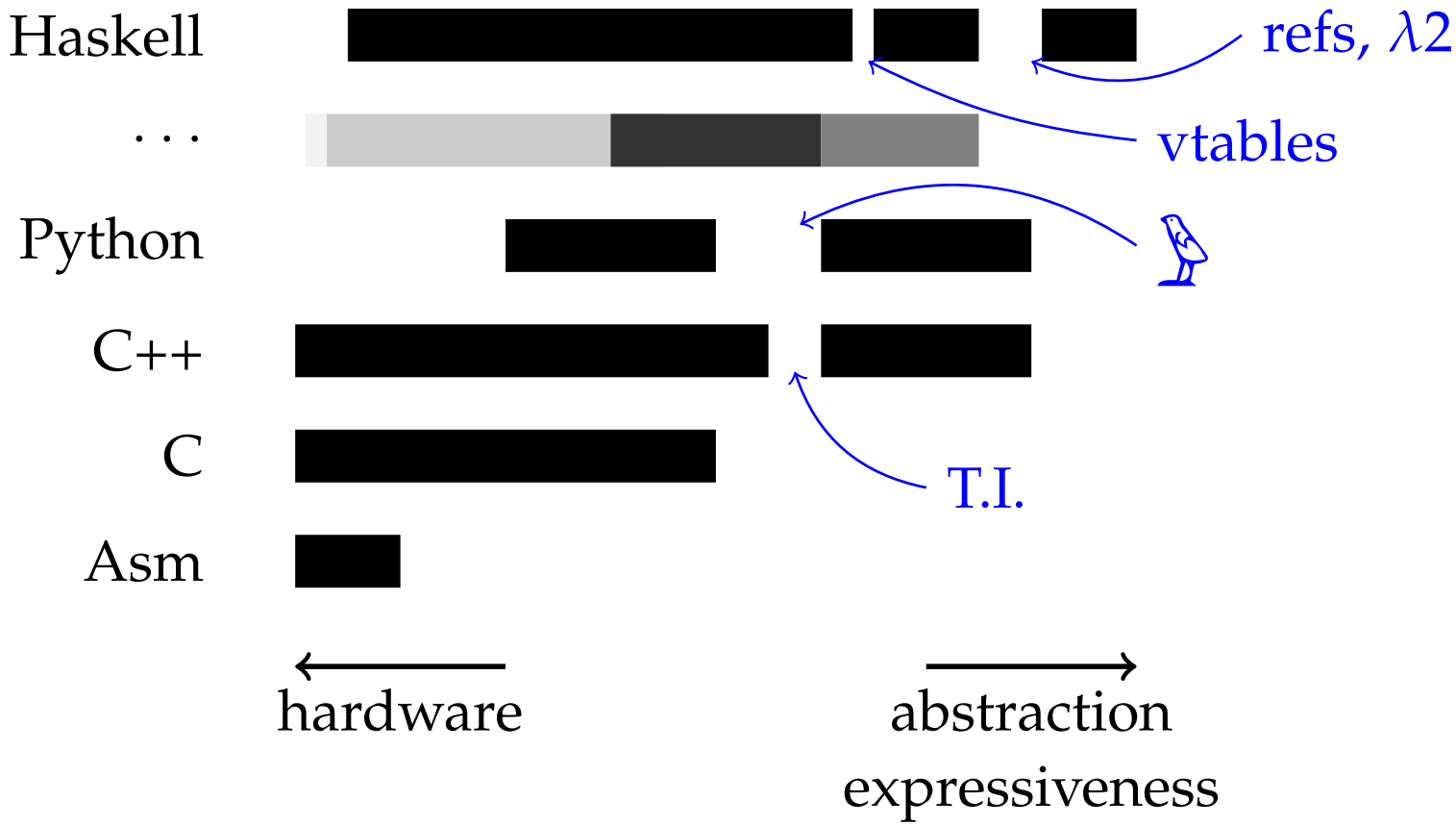
2016

Motivation

Feature sets of current languages, by hardware abstraction:

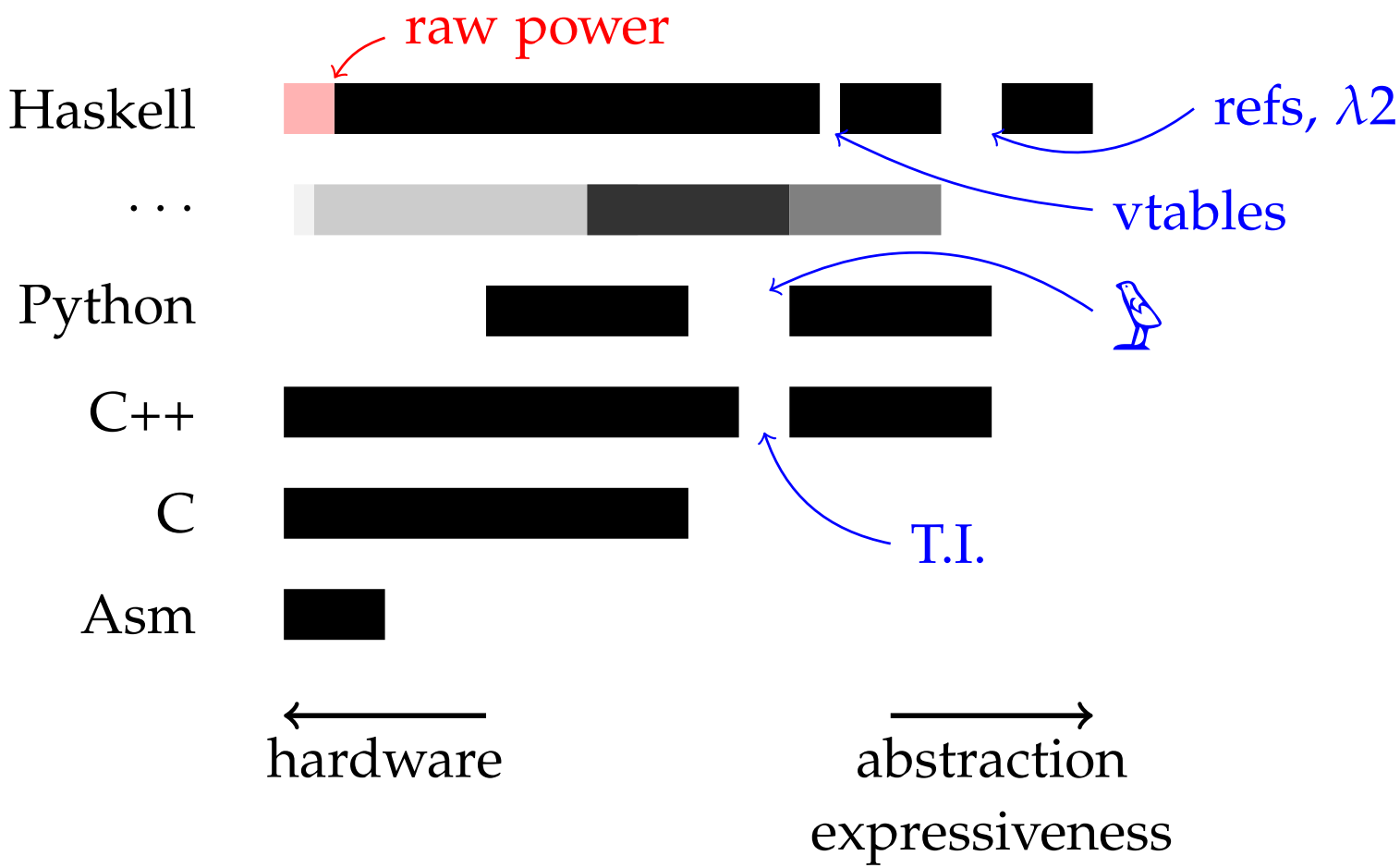
Motivation

Feature sets of current languages, by hardware abstraction:



Motivation

Feature sets of current languages, by hardware abstraction:



Background

- Processing of pure functional languages is easy
  - side effects are clearly specified
  - no explicit structuring of code
  - type safety, high levels of abstraction for free
- Low-level ("system") languages could benefit from this
  - resulting code could get faster
    (high-level optimizations, auto-parallelization)
  - increased viability of automatic invariant derivation
    (more optimizations, verification etc.)

Related achievements

- Allocation/placement-based performance speedups in functional languages:

  Guy E. Blelloch and Robert Harper. Cache efficient functional algorithms. *Communications of the ACM*, 58(7):101–108, 2015.

- Generative approach to *portable* auto-parallelization:

  Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules. 2015.

- λ–calculus-based function lifting, dropping, specialization, generalization, ...

  Olivier Danvy and Ulrik P Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. ACM, 1997.

LLFP examples

List processing:

```
printList p =
  when p {
    List val nxt <- read p;
    print val;
    printList nxt;
  };
```

Fun with inference:

```
new :: IO (Ptr a)
```

Portably parallel (crude) code from Steuwer et al. (2015):

```
scale a = map (*a)

dot x y =
   (map (*) . foldl (+) 0)
     (zip x y)

gemv mtx x y a b =
   map (+) $ zip
   (map (dot x . scale a) mtx)
   (scale b y)
```

(gemv computes $\alpha M\vec{x} + \beta\vec{y}$)

Problem

Current pure[1], lazily-evaluated functional languages require automatic memory management.

- A.M.M. $\rightarrow$garbage collection
- direct conflict with systems programming:
  - fitting on stack
  - realtime requirements
  - placement

[1] Haskell etc.

Approach

A new Haskell-like language that replaces features from F.L. that require automatic memory management:

1. Recursive types (e.g. lists, trees, ...)
2. Unbounded recursion
3. Lazy evaluation (also covers partially applied functions)

The code is compiled to a standard standalone executable with no runtime support (equivalent to the output of C compiler).

Recursive types

$$\text{data } \mathbf{Tree}\ a = \mathbf{Node}\ (\mathbf{Tree}\ a)(\mathbf{Tree}\ a)|\mathbf{Nil}$$

Direct implications:

- uncertainity of object size
- uncertainity of deallocation point

Solutions:

- "The C++ way."
  - replacement by pointed types
  - STL style of hiding complexity (even GC)
- Another possibility: linear types (from linear logic).

Unbounded recursion

Common effective solution: Tail call elimination.
The rest:

- Problem: stack overflows.

$\Rightarrow$ Warning on unbounded non-tail recursion.

N.B. Recursion elimination is quite simple (continuation-passing-style can be humanized by monads[2]).

[2] e.g. Russell O'Connor. Continuation Passing Style for Monads. 2007. `http://r6.ca/blog/20071028T162529Z.html`

Lazy evaluation

The most problematic part. Toy example:

$$x \leftarrow \text{readInt}$$
$$\text{map } (+x) \text{ someArray}$$

Possible approaches:

- Inline 'map'.
  (termination is uncertain, danger of code duplication)
- Generate a *thunk* that describes $(+x)$, pass it to 'map'.
  - Thunk stores the environment (x)
  - 'map' can *force* the thunk after it gets enough arguments.
  - Haskell allocates thunks dynamically
    run-time checks added to recognize thunks vs. values

Lazy evaluation

Our approach: Place the static thunks on stack

1. For each variable occurence, determine thunk status&size at compile-time
2. Specialize functions to accept such thunks (overloading)

All required information is derived by type inference!

Example

$$
\begin{aligned}
f :: &\quad \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int} \\
g :: &\quad \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int} \to \mathbf{Int} \\
\\
h\ b\ i = &\quad \mathbf{if}\ b\ \mathbf{then}\ f\ i \\
&\quad \mathbf{else}\ g\ i\ (i+1)
\end{aligned}
$$

- Haskell: $(h\ \text{true}\ 1) :: \mathbf{Int} \to \mathbf{Int}$
- our target type: $\mathbf{thunk}(\{\mathbf{Int}\}, \mathbf{Int} \to \mathbf{Int})$
  (contents: function pointer, stored argument)

Hindley-Milner type inference (original)

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \text{Var} \qquad \frac{\Gamma \vdash e : \tau \quad \sigma \in \text{instances}_\Gamma(\tau)}{\Gamma \vdash e : \sigma}\ \text{Inst}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \text{close}(\tau) \vdash y : \sigma}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ y : \sigma}\ \text{Let}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x.e) : \sigma \to \tau)}\ \text{Abs}$$

$$\frac{\Gamma \vdash e : \sigma \to \rho \quad \Gamma \vdash x : \sigma}{\Gamma \vdash (e\ x) : \rho}\ \text{App}$$

(omitted: parametric overloading as presented by Kaes, recursive let)

H-M with thunks

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \text{Var} \qquad \frac{\Gamma \vdash e : \tau \quad \sigma \in \text{instances}_\Gamma(\tau)}{\Gamma \vdash e : \sigma}\ \text{Inst}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \text{close}(\tau) \vdash y : \sigma}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ y : \sigma}\ \text{Let}$$

$$\frac{\Gamma, x_{1,2,\ldots,n} : \sigma_{1,2,\ldots,n} \vdash e : \tau}{\Gamma \vdash (\lambda x_{1,2,\ldots,n}.e) : \textbf{thunk}(\{\varnothing\}, \sigma_1 \to \sigma_2 \to \cdots \to \sigma_n \to \tau)}\ \text{Abs}$$

$$\frac{\Gamma \vdash e : \textbf{thunk}(\Phi, \sigma \to \rho) \quad \Gamma \vdash x : \sigma}{\Gamma \vdash (e\ x) : \textbf{thunk}(\Phi \parallel \sigma, \rho)}\ \text{App}$$

$$\frac{\Gamma \vdash e : \textbf{thunk}(\Phi, \rho) \quad \rho \in \mathcal{R}}{\Gamma \vdash e : \rho}\ \text{Force}$$

H-M with thunks

Robinson unification on thunks:

$$\begin{aligned}\text{Unify}(\mathbf{thunk}(A,\sigma),\mathbf{thunk}(B,\tau)) \Leftarrow & \text{Unify}(\sigma,\tau),\\ & N = A \cup B,\\ & \text{Subst}(A,N),\\ & \text{Subst}(B,N).\end{aligned}$$

Derivation example:

$$\dfrac{\dfrac{\Gamma \vdash A : \mathbf{thunk}(\{\sigma,\varnothing\},\alpha \to \beta) \qquad \Gamma \vdash B : \alpha}{\Gamma \vdash AB : \mathbf{thunk}(\{\sigma\alpha,\alpha\},\beta)}\,\text{App}}{\Gamma \vdash AB : \beta}\,\text{Force}$$

Set content: types of all possible thunk-evaluating functions.

Results

- New compiler is surprisingly simple
- Thunk construction gives new possibilities for code translation.
- Possibility of work duplication must be mitigated.

Practical performance on-par with equivalent programs in C++, Haskell shows some overhead.

Conclusions

- No serious drawbacks of low-level functional programming found.
- Despite simplicity, the compiler is able to handle very expressive, complicated constructions.
- Several new language-theoretic problems surfaced.

Current WIP: Type system with typeclass-based overloading (like in Haskell).

Future work

- Emulate some popular C++ features:
  - Variable-holding semantics from C++
    (better destructors than `bracket`)
  - Run-time polymorphism
    (already possible with thunks!)
- Functional approaches to parallelization on bare hardware
- Current speculations:
  - Optimization of all non-deterministic operation
    (thunks, lambda dropping, ...)
  - Exploit language simplicity to aid compaction
    ("uninlining"; related: macro compression)

Thank you for your attention.
(questions?)