Let’s compile some Haskell!

A very rough overview of Haskell compilation in GHC

1. Parsing
2. Typechecking
3. Desugaring (conversion of the language to Core)
4. Optimization (“simplifier”)
5. Conversion to CPS and STG to solve laziness, more optimization
6. Conversion to Cmm or LLVM, even more optimization
7. Building of object code
8. Linking with RTS and other libraries

There are also other ways! See Hugs, MicroHs, ...

A very rough overview of Haskell compilation in GHC

1. Parsing
2. Typechecking
3. Desugaring (conversion of the language to Core)
4. Optimization (“simplifier”)
5. Conversion to CPS and STG to solve laziness, more optimization
6. Conversion to Cmm or LLVM, even more optimization
7. Building of object code
8. Linking with RTS and other libraries

There are also other ways! See Hugs, MicroHs, …

Simon Peyton Jones & Simon Marlow

How do we infer and check types in functional code?

Difference from C-like languages: ‘simulating the evaluation on types’ doesn’t tell us much:

- literals may have any type; defaulting and “converting later just in case” can’t be done
- sometimes we need to guess the type of function parameter
- forward inference fails in presence of recursion

In the following slides we briefly show Hindley-Damas-Milner (H-M) type inference system and its application in Haskell.

We will need a lot of lambda calculus

Lambda calculus is attributed to Alonzo Church (1930-ish).

$V ::= a \mid b \mid c \mid \cdots \mid x_\infty$	variables are marked by letters
$\Lambda ::= V$	λ-terms are either variables
$\mid \Lambda\Lambda$	or λ-terms applied to λ-terms
$\mid (\lambda V.\Lambda)$	or λ-functions that bind an argument variable

Surprisingly,

- The system is T-complete
- Thanks to the simplicity it makes a great model for “human” programming languages.

Lambda calculus — working with terms

- α-equivalence
  $(\lambda x. \ldots x \ldots) = (\lambda y. \ldots y \ldots)$
- β-reduction
  $(\lambda x.M)y = M[x := y]$
- η-conversion
  $(\lambda x.Mx) = M$

Lambda calculus evaluates by β-reduction

$$
\begin{aligned}
(\lambda x.\lambda y.y\ x\ x)\ a\ f &= (\lambda y.y\ a\ a)\ f \\
&= f\ a\ a
\end{aligned}
$$

Example from Haskell:

$$
\begin{aligned}
\text{flip}\ (-)\ a\ b &= (\lambda f.\lambda x.\lambda y.f\ y\ x)\ (-)\ a\ b \\
&= (\lambda x.\lambda y.(-)\ y\ x)\ a\ b \\
&= (\lambda y.(-)\ y\ a)\ b \\
&= (-)\ b\ a
\end{aligned}
$$

Example that loops forever:

$$
\begin{aligned}
\omega\omega &= (\lambda x.x\ x)\ (\lambda x.x\ x) \\
&= (\lambda x.x\ x)\ (\lambda x.x\ x) \\
&= \ldots
\end{aligned}
$$

Lambda calculus does not actually have any values?

```
#haskell
20:30 < Ariakenom> lambda calculus has both
                   functions and values,
                   because functions
                   are values
```

Alonzo Church

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

$\mathbf{Z?} = (\lambda n.n(KF)T), \mathbf{incr} = \mathbf{add\,1} = (\lambda bfx.f(bfx)) = (\lambda bfx.bf(fx))$

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

$\mathbf{Z?} = (\lambda n.n(KF)T), \mathbf{incr} = \mathbf{add}\,\mathbf{1} = (\lambda bfx.f(bfx)) = (\lambda bfx.bf(fx))$

$\mathbf{pair} = (\lambda lrx.xlr), \mathbf{left} = (\lambda x.x\mathbf{T}), \mathbf{right} = (\lambda x.x\mathbf{F})$

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

$\mathbf{Z?} = (\lambda n.n(KF)T), \mathbf{incr} = \mathbf{add}\,\mathbf{1} = (\lambda bfx.f(bfx)) = (\lambda bfx.bf(fx))$

$\mathbf{pair} = (\lambda lrx.xlr), \mathbf{left} = (\lambda x.x\mathbf{T}), \mathbf{right} = (\lambda x.x\mathbf{F})$

$\mathbf{decr} = (\lambda n.\mathbf{right}(n\ \mathbf{incr2}\ (\mathbf{pair}\ \mathbf{0}\ \mathbf{0}))),$
$\mathbf{incr2} = (\lambda p.\mathbf{pair}\ (\mathbf{incr}\ (\mathbf{left}\ p))\ (\mathbf{if}\ (\mathbf{Z?}\ (\mathbf{left}\ p))\ \mathbf{0}\ (\mathbf{incr}\ (\mathbf{right}\ p))))$

$I = (\lambda x.x), K = (\lambda xy.x), K' = (\lambda xy.y),$
$\mathbf{T} = K, \mathbf{F} = K', \mathbf{if} = (\lambda btf.btf) = I$

$\mathbf{0} = (\lambda fx.x) = \mathbf{F}, \mathbf{1} = (\lambda fx.fx) = I, \mathbf{2} = (\lambda fx.f(fx)), \mathbf{n} = (\lambda fx.f^n x),$
$\mathbf{add} = (\lambda abfx.af(bfx)), \mathbf{mul} = (\lambda abf.a(bf)), \mathbf{exp} = (\lambda ab.ba)$

$\mathbf{Z?} = (\lambda n.n(KF)T), \mathbf{incr} = \mathbf{add\,1} = (\lambda bfx.f(bfx)) = (\lambda bfx.bf(fx))$

$\mathbf{pair} = (\lambda lrx.xlr), \mathbf{left} = (\lambda x.x\mathbf{T}), \mathbf{right} = (\lambda x.x\mathbf{F})$

$\mathbf{decr} = (\lambda n.\mathbf{right}(n\ \mathbf{incr2}\ (\mathbf{pair\ 0\ 0}))),$
$\mathbf{incr2} = (\lambda p.\mathbf{pair}\ (\mathbf{incr}\ (\mathbf{left}\ p))\ (\mathbf{if}\ (\mathbf{Z?}\ (\mathbf{left}\ p))\ \mathbf{0}\ (\mathbf{incr}\ (\mathbf{right}\ p))))$

Recursion is possible without using your own definition:

$\mathbf{facF} = (\lambda rn.\mathbf{if}(\mathbf{Z?}\ n)\ \mathbf{1}\ (\mathbf{mul}\ n\ (r(\mathbf{decr}\ n))))$

$\mathbf{fac} = Y\ \mathbf{facF}, Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))), \infty = (\lambda fx.Yfx) = Y$

Fixed point theorem enables recursion in λ-calculus

Theorem: $(\forall F \in \Lambda)(\exists X \in \Lambda) \quad X = F(X)$

Proof:

Fixed point theorem enables recursion in λ-calculus

Theorem: $(\forall F \in \Lambda)(\exists X \in \Lambda) \quad X = F(X)$

Proof:

- let's make a gadget: $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$
- take $X = YF$
- thus $YF = F(YF)$ by β-reduction
- in turn $X = FX$. ✌

Fixed point theorem enables recursion in λ-calculus

Theorem: $(\forall F \in \Lambda)(\exists X \in \Lambda) \quad X = F(X)$

Proof:

- let's make a gadget: $Y = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$
- take $X = YF$
- thus $YF = F(YF)$ by β-reduction
- in turn $X = FX$. ✌

Corrolary: $YF = F(YF) = F(F(YF)) = F^{\infty}(YF)$.

Fixed point combinator

Y is traditionally referred to as 'fixed point combinator' because of the similarity to fixed-point theorems from real calculus.

- Calculus: find x such that after evaluating $f(x)$ we get x, thus $x = f(x)$.
- λ: find X such that after evaluating X we get $F(X)$, thus $X = F(X)$.

Enumerability theory calls it a 'while-loop':

- $F^{\infty}(X)$ may loop forever
- a condition in F may stop the loop anytime

N.B.: we don't really need to use Y for writing factorials! It's sufficient to use the "bounded" loops provided by the numerals.

Fixpoint exists in Haskell, sometimes it's even useful.

let $fix\ f = f\ (fix\ f)$

Exercise — what does this do?:

- $fix\ (\lambda f\ i \rightarrow i : f\ (i + 1))\ 1$
- $fix\ (1:)$
- $fix\ show$
- $fix\ (scanl\ (+)\ 0 \cdot (1:))$
- $fix\ error$

How to finish evaluating anything, ever

If a λ-term admits a *normal form* ("sufficiently evaluated one"), it is always reachable by evaluating the **left-most redex**.

- $(\lambda x.xx)\ (\lambda x.xx)$
- $K'\ ((\lambda x.xx)\ (\lambda x.xx))(\lambda x.x)$

This precise evaluation strategy is called **lazy evaluation**.

Technically, we don't even need variables

Closed terms are called combinators (F, K, S, I, ...)

Theorem (SKI calculus): Any λ-term can be equivalently written as an expression that consists only of combinators

$$S = \lambda xyz.xz(yz)$$
$$K = \lambda xy.x$$

Examples:

$$I = SKK$$
$$K' = SK$$
$$F = S(K(SI))K \qquad \texttt{--flip}$$
$$Y = S(K(SII))(S(S(KS)K)(K(SII)))$$
$$\vdots$$

Combinator calculus is point-free

Similarly, we can get rid of abstractions in any Haskell term, getting a “point-free form”.[2]

Simple conversions:

$\lambda x \to x$	id
$\lambda x \to a$	$const\ a$
$\lambda x \to a\ x$	a
$\lambda x \to a\ (b\ x)$	$a \cdot b$
$\lambda x \to (a\ x)\ b$	$flip\ a\ b$
$\lambda x \to (a\ x)\ (b\ x)$	$a \langle * \rangle b$
$\lambda x \to \lambda y \to a$	recursively
x nested	split scope with new abstraction

[2] point-less form

Can we statically see what a given λ-term does?

Idea: let's describe useful types and see what happens.

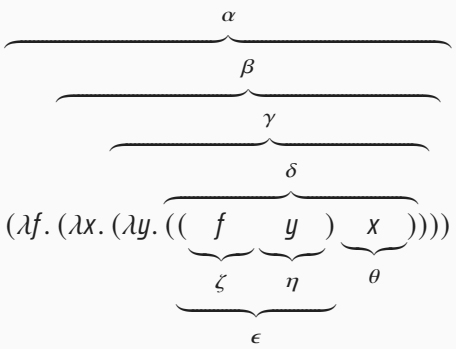
Simply-typed lambda calculus (STLC):

$$
\begin{array}{rll}
V_T & ::= \alpha \mid \beta \mid \gamma \mid \cdots \mid \tau_\infty & \text{type-level variables} \\
T & ::= V_T & \text{simple type} \\
& \mid T \to T & \text{function type}
\end{array}
$$

Usual signage: $\Gamma \vdash M : \tau$

Examples: $\vdash I : \tau \to \tau$ $\quad \vdash K : \alpha \to \beta \to \alpha$ $\quad \vdash \mathbf{n} : (\sigma \to \sigma) \to \sigma \to \sigma$
$\vdash \mathbf{if} : (\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \to \alpha$ $\quad \{x : \alpha \to \beta, y : \alpha\} \vdash xy : \beta$

STLC types are constrained by equations



We solve a system of symbolic equations:

$\alpha = \zeta \to \beta$	...looking up type of *f*
$\beta = \theta \to \gamma$	...looking up *x*
$\gamma = \eta \to \delta$	...*y*
$\epsilon = \theta \to \delta$	...applying $(fy)x$ yields δ
$\zeta = \eta \to \epsilon$	...applying *fy* yields ϵ

(NB.: Each equation corresponds to a single syntactic construction, here 3 abstractions and 2 applications. Variable scoping and lookups are done “manually” here.)

Type equations get solved by unification and substitution

$$\underline{\alpha} = \zeta \to \beta$$
$$\beta = \theta \to \gamma$$
$$\gamma = \eta \to \delta$$
$$\epsilon = \theta \to \delta$$
$$\zeta = \eta \to \epsilon$$

Substitutions:

Type equations get solved by unification and substitution

$$\underline{\alpha} = \zeta \to \theta \to \gamma$$

$$\gamma = \eta \to \delta$$
$$\epsilon = \theta \to \delta$$
$$\zeta = \eta \to \epsilon$$

Substitutions: $\beta := \theta \to \gamma$

Type equations get solved by unification and substitution

$$\underline{\alpha} = \zeta \to \theta \to \eta \to \delta$$

$$\epsilon = \theta \to \delta$$
$$\zeta = \eta \to \epsilon$$

Substitutions: $\beta := \theta \to \gamma \quad \gamma := \eta \to \delta$

Type equations get solved by unification and substitution

$$\underline{\alpha} = \zeta \to \theta \to \eta \to \delta$$

$$\zeta = \eta \to \theta \to \delta$$

Substitutions: $\beta := \theta \to \gamma \quad \gamma := \eta \to \delta \quad \epsilon := \theta \to \delta$

Type equations get solved by unification and substitution

$$\underline{\alpha} = (\eta \to \theta \to \delta) \to \theta \to \eta \to \delta$$

Substitutions: $\beta := \theta \to \gamma$ $\gamma := \eta \to \delta$ $\epsilon := \theta \to \delta$ $\zeta := \eta \to \theta \to \delta$

Robinson unification solves symbolic equations

What about more complex cases? $\alpha \to (\beta \to \gamma) = \delta \to \alpha$?

Robinson unification algorithm for STLC

Equation	What to do?
$\alpha = \alpha$	all good, continue with next equation
$\alpha = \ldots \alpha \ldots$	fail (solution would create infinite type; e.g. in $\lambda x.xx$)
$\alpha = X$	check $\alpha \notin X$, substitute all $\alpha := X$, continue
$X = \alpha$	flip to previous case
$A \to B = C \to D$	add equations $A = C$ and $B = D$ to the queue, continue

STLC is usually described by natural deduction rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ (Var)}$$

$$\frac{\Gamma \vdash M : \pi \to \rho \quad \Gamma \vdash N : \pi}{\Gamma \vdash MN : \rho} \text{ (App)}$$

$$\frac{\Gamma \cup \{v : \pi\} \vdash M : \rho}{\Gamma \vdash (\lambda v.M) : (\pi \to \rho)} \text{ (Abs)}$$

The rules are sometimes dubbed Axiom, $\to$-elimination and $\to$-introduction.

STLC guarantees that typed programs finish!

Weak normalization property of λ-calculus:

$$(\forall M \in \Lambda) \qquad M : \tau \implies M \text{ has a normal form}$$

Unfortunately:

- Reverse implication does not hold.
- As a common computing tragedy, most useful programs don't have a STLC type, but still have a normal form "by coincidence".

STLC guarantees that typed programs finish!

Weak normalization property of λ-calculus:

$$(\forall M \in \Lambda) \qquad M : \tau \implies M \text{ has a normal form}$$

Unfortunately:

- Reverse implication does not hold.
- As a common computing tragedy, most useful programs don't have a STLC type, but still have a normal form "by coincidence".
- There are type systems with **strong normalization** (such as $\lambda^{\rightarrow}_{\cap}$) where exactly all terminating programs have a type, but the type inference is undecidable there.
  In enumerability theory, this corresponds to the differences between primitive, total and partial recursive functions.

STLC is not sufficiently polymorphic

Normally we want to be able to re-use polymorphic things for many different types.

Most primitively, this should work:

$$print\ (id\ \texttt{"hello"}, id\ 12345)$$

In STLC, using $I : \alpha \to \alpha$ twice will cause its type to unify with both $\mathsf{String} \to \mathsf{String}$ and $\mathsf{Int} \to \mathsf{Int}$, and the deduction will fail.

SystemF allows polymorphic types

Girard (1972) and independently Reynolds (1974) introduced polymorphic lambda terms ('second order calculus' or '$\lambda 2$' or SystemF):

$$T ::= V_T \mid T \to T \mid \forall V_T.T$$

Additional rules added to STLC:

$$\frac{\Gamma \vdash M : \sigma \quad \alpha \notin \Gamma}{\Gamma \vdash M : (\forall \alpha.\sigma)} \text{ (Generalization)}$$

$$\frac{\Gamma \vdash M : (\forall \alpha.\sigma)}{\Gamma \vdash M : \sigma[\alpha := \tau]} \text{ (Instantiation)}$$

"Instantiation" can be viewed as "specialization".

Jean-Yves Girard & John C. Reynolds

SystemF is a bit too strong

Pros:

- $I : \forall\alpha.\alpha \to \alpha$, can be used multiple times!
- $(\lambda x.xx) : (\forall\beta.(\forall\alpha.\alpha) \to \beta)$

Cons:

- $(\lambda x.xx) : (\forall\beta.(\forall\alpha.\alpha) \to (\beta \to \beta))$
- $(\lambda x.xx) : (\forall\alpha.\alpha) \to (\forall\alpha.\alpha)$
- inference is undecidable

  ...

Obviously, the use of generalization is a bit too much generous. Let's try a more restricted form.

Hindley-Damas-Milner inference allows limited polymorphism

To tame the polymorphism, we introduce a new construction **let** $v = d$ **in** e, the result of which is a polymorphic v that can be used in e.

Type generalization can thus be limited to the “top level”:

$$\Lambda ::= V \mid \Lambda\Lambda \mid (\lambda V.\Lambda) \mid \textbf{let } V = \Lambda \textbf{ in } \Lambda$$

$$\mathcal{M} ::= V_T \mid \mathcal{M} \to \mathcal{M} \qquad \text{(monotypes)}$$

$$\mathcal{P} ::= \mathcal{M} \mid \forall V_T.\mathcal{P} \qquad \text{(polytypes)}$$

Instantiation and generalization in H-M is driven by syntax

$$\frac{x : \tau \in \Gamma \quad \tau \in \mathcal{M}}{\Gamma \vdash x : \tau} \text{ (Var-Mono)}$$

$$\frac{x : \tau \in \Gamma \quad \tau \in \mathcal{P}}{\Gamma \vdash x : \text{instantiate}(\tau)} \text{ (Var-Poly)}$$

$$\frac{\Gamma \vdash a : \pi \to \rho \quad \Gamma \vdash b : \pi}{\Gamma \vdash a\, b : \rho} \text{ (App)}$$

$$\frac{\Gamma \cup \{v : \pi\} \vdash e : \rho}{\Gamma \vdash (\lambda v.e) : (\pi \to \rho)} \text{ (Abs)}$$

$$\frac{\Gamma \vdash d : \tau \quad \Gamma \cup \{v : \text{quantify}(\tau)\} \vdash e : \sigma}{\Gamma \vdash (\textbf{let}\ v = d\ \textbf{in}\ e) : \sigma} \text{ (Let-Gen)}$$

H-M computation is similar to STLC

- Variables captured by **let** receive the most generic possible polytype ("principial type")
- When using a polymorphic variable, we instantiate it to a monomorphic one, using *fresh* type variables (not used elsewhere).

Example: **let** *I* $= (\lambda x.x)$ **in** *I I*

- $(\lambda x.x) : \alpha \to \alpha$
- $I : \forall\alpha.\alpha \to \alpha$
- Second *I*: $\beta \to \beta$
- First *I*: $\gamma \to \gamma$, consequently $(\beta \to \beta) \to (\beta \to \beta)$
- Whole expression: $\beta \to \beta$

There's no polymorphism without instantiation!

$$\mathit{id}\ \mathit{id} :: a \to a$$

...but:

$$(\lambda x \to x\ x)\ \mathit{id}$$

Occurs check: cannot construct the infinite type: $t \sim t \to t$

Type classes are added on top of H-M framework

H-M support for typeclasses:

- types of classes and instances are collected before typechecking
- polytypes and contexts become qualified
  a qualification is a set of predicates assigned to the type variables: $(\mathrm{Ord}\ k, \mathrm{Num}\ a)$
- typechecking process is extended:
  - substitutions are also applied to the qualification
  - known types in predicates must satisfy a single instance (This works a lot like Prolog.)

Type classes are added on top of H-M framework

H-M support for typeclasses:

- types of classes and instances are collected before typechecking
- polytypes and contexts become qualified
  a qualification is a set of predicates assigned to the type variables: (Ord k, Num a)
- typechecking process is extended:
  - substitutions are also applied to the qualification
  - known types in predicates must satisfy a single instance (This works a lot like Prolog.)
- there are some new issues: ...which is a common price for added features
  - recursive inference becomes undecidable
  - sometimes the context remains ambiguous on which instance to select
    $f = show \cdot read$

✎Mark P. Jones, “Typing Haskell In Haskell.” `http://web.cecs.pdx.edu/~mpj/thih/`

Common issue: Recursive polymorphism is complicated

Inference will sometimes fail on polymorphically recursive definitions!

```
data FullTree a = Nil | Node a (FullTree (a, a))

  treemap f Nil = Nil
  treemap f (Cons x xs) =
    Cons (f x) (treemap (λ(a, b) → (f a, f b)) xs)
```

Error: cannot construct the infinite type...

As the simplest fix, you can spoil the solution to the compiler:

```
treemap :: (a → b) → FullTree a → FullTree b
```

Defaulting resolves some ambiguities

$$f = show \cdot read$$
$$f :: \text{String} \to \text{String}$$

...but what actually happens in between?

Similarly:

$$main = print\ 5$$

Do we get an Int, Float, Integer, or something different?

Defaulting: Compiler will assume the first type from *default list* that fits.

Example settings: **default** (Integer, Double)

Turn off the defaulting for a module: **default** ()

Internally, ambiguities stem from leaking internal predicates

Example derivation with $f\ x = show\ (read\ x)$:

- $read :: \text{Read}\ a \Rightarrow \text{String} \to a$
- $x :: \text{String}$
- $(read\ x) :: \text{Read}\ a \Rightarrow a$
- $show :: \text{Show}\ a \Rightarrow a \to \text{String}$
- $show\ (read\ x) :: (\text{Read}\ a, \text{Show}\ a) \Rightarrow \text{String}$
- $f :: (\text{Read}\ a, \text{Show}\ a) \Rightarrow \text{String} \to \text{String}$

The ambiguities exactly correspond to qualified variables not used in the type.

Ambiguities are *resolved* by adopting the first type from the **default**-list that matches all predicates (or by throwing an error).

Modern Haskell type system is a bit more complex

- multi-parameter type classes
- DataKinds, “dependent” types
- explicit type equality
- type families
- linear types
- inference is essentially a huge constraint solver
- ...

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : \_$

Example of using STLC: Syntax-driven way

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_1$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : \tau_2 \to \tau_3$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : \tau_2 \to \tau_4 \to \tau_5$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : \tau_2 \to \tau_4 \to \tau_6 \to \tau_7$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : \tau_2 \to \tau_4 \to \tau_6 \to \tau_9$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : \tau_2 \to \tau_8 \to \tau_6 \to \tau_9$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$

Example of using STLC: Syntax-driven way

$\mathbf{flip} = (\lambda fxy.(fy)x), \quad \mathbf{flip} : \tau_2 \to \tau_8 \to \tau_6 \to \tau_9$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : \tau_2 \to \tau_8 \to \tau_{10} \to \tau_9$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash y : \tau_6$	$\tau_6 = \tau_{10}$

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : (\tau_{10} \to \tau_{11}) \to \tau_8 \to \tau_{10} \to \tau_9$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash y : \tau_6$	$\tau_6 = \tau_{10}$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_2$	$\tau_2 = \tau_{10} \to \tau_{11}$

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : (\tau_{10} \to \tau_8 \to \tau_9) \to \tau_8 \to \tau_{10} \to \tau_9$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash y : \tau_6$	$\tau_6 = \tau_{10}$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_2$	$\tau_2 = \tau_{10} \to \tau_{11}$

Example of using STLC: Syntax-driven way

$\textbf{flip} = (\lambda fxy.(fy)x), \quad \textbf{flip} : (\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$

Axiom	Derivation	Substitution
—	$\vdash (\lambda fxy.(fy)x) : \tau_1$	—
(Abs)	$\{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_3, \vdash (\lambda fxy.(fy)x) : \tau_2 \to \tau_3$	$\tau_1 = \tau_2 \to \tau_3$
(Abs)	$\{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_5, \{f : \tau_2\} \vdash (\lambda xy.(fy)x) : \tau_4 \to \tau_5$	$\tau_3 = \tau_4 \to \tau_5$
(Abs)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy)x : \tau_7, \{f : \tau_2, x : \tau_4\} \vdash (\lambda y.(fy)x) : \tau_6 \to \tau_7$	$\tau_5 = \tau_6 \to \tau_7$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash (fy) : \tau_8 \to \tau_9 \wedge x : \tau_8$	$\tau_7 = \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash x : \tau_4$	$\tau_4 = \tau_8$
(App)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_{10} \to \tau_{11} \wedge y : \tau_{10}$	$\tau_{11} = \tau_8 \to \tau_9$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash y : \tau_6$	$\tau_6 = \tau_{10}$
(Var)	$\{f : \tau_2, x : \tau_4, y : \tau_6\} \vdash f : \tau_2$	$\tau_2 = \tau_{10} \to \tau_{11}$

Example of using STLC: French way

Sub-expression	Rule	Type	Context	Equation
$(\lambda fxy.(fy)x)$	–	τ_1	–	–
$(\lambda xy.(fy)x)$	Abs	τ_2	$f : \tau_3$	$\tau_1 = \tau_3 \to \tau_2$
$(\lambda y.(fy)x)$	Abs	τ_4	$x : \tau_5$	$\tau_2 = \tau_5 \to \tau_4$
$(fy)x$	Abs	τ_6	$y : \tau_7$	$\tau_4 = \tau_7 \to \tau_6$
$(fy)x$	App	τ_6	$fy : \tau_8$	$\tau_8 = \tau_5 \to \tau_6$
fy	App	τ_8	–	$\tau_3 = \tau_7 \to \tau_8$

We solve by substitution:

$$\begin{aligned}\tau_1 &= \tau_3 \to \tau_2 \\ &= (\tau_7 \to \tau_8) \to \tau_5 \to \tau_4 \\ &= (\tau_7 \to \tau_5 \to \tau_6) \to \tau_5 \to \tau_7 \to \tau_6 \\ &= (\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma\end{aligned}$$

Core

After the code is typechecked, it is rewritten to Core.

```
data Expr b = Var   b
            | Lit   Literal
            | App   (Expr b) (Arg b)
            | Lam   b (Expr b)
            | Let   (Bind b) (Expr b)
            | Case  (Expr b) b Type [Alt b]
            | Cast  (Expr b) Coercion
            | Tick  (Tickish Id) (Expr b)
            | Type Type
            | Coercion Coercion
```

(`CoreSyn.hs:255-266`)

Core is a simple pure lazy IR in GHC

Typeclasses do not reach Core!

- type qualifications become function parameters
- the qualification values are “dictionary functions” (basically vtables, but nicer)

Core benefits:

- simple code admits simple optimizations
  - DCE
  - Inlining
  - Specializations
- it may be typechecked again to validate the compiler functionality

Nested functions are compiled out by lifting

Neither of STG, Cmm and object code handle variable scopes. Variable lifting is a simple transformation that makes the scope explicit:

$$\begin{aligned} &solutions\ a\ b\ c = map\ (/(2 * a))\ [-b + sD, -b - sD] \\ &\quad \textbf{where}\ sD = sqrt\ (b * b - 4 * a * c) \end{aligned}$$

Lifting the argument variables:

$$\begin{aligned} &solutions\ a\ b\ c = map\ (/(2 * a))\ [-b + sD\ a\ b\ c, -b - sD\ a\ b\ c] \\ &\quad \textbf{where}\ sD\ a\ b\ c = sqrt\ (b * b - 4 * a * c) \end{aligned}$$

Floating the definition out:

$$\begin{aligned} &sD\ a\ b\ c = sqrt\ (b * b - 4 * a * c) \\ &solutions\ a\ b\ c = map\ (/(2 * a))\ [-b + sD\ a\ b\ c, -b - sD\ a\ b\ c] \end{aligned}$$

The need for return stack is eliminated by CPS conversion

Conversion to Continuation-Passing-Style:

$$sD\ a\ b\ c\ cont = cont\ (sqrt\ (b * b - 4 * a * c))$$

$$solutions\ a\ b\ c\ cont = sD\ a\ b\ c\ \$\ \lambda d \rightarrow$$
$$map\ (/(2 * a))\ [-b + d, -b - d]\ cont$$

$$map\ \_\ [\,]\ cont = cont\ [\,]$$
$$map\ f\ (x : xs)\ cont = f\ x\ \$\ \lambda x' \rightarrow$$
$$map\ f\ xs\ \$\ \lambda xs' \rightarrow$$
$$cont\ (x' : xs')$$

(We will pretend that *sqrt* and other numeric operations are ***primitive***)

- CPS'd function are always tail calls and never return.
- The continuation often becomes just a code pointer.
- IO becomes a huge continuation that keeps passing the RealWorld around.
- Unused and unreachable branches are GC'd together with their data.

STG is impure strict IR in GHC

Lifted&CPS'd Core is rewritten to STG, which is a "VM" for executing Haskell.

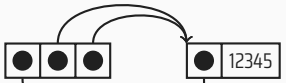
STG contains:

- registers
- heap
  - data
  - functions, closures, thunks
  - indirections
  - continuations that were ejected from stack
- continuation stack
  - Haskell has no data stack!
  - used only as a specialized fast storage for "return pointers" and closures

All heap values are connected by pointers, forming a huge graph.

STG objects are identified by the code that evaluates them

let $x = 12345$ **in** $x + x$



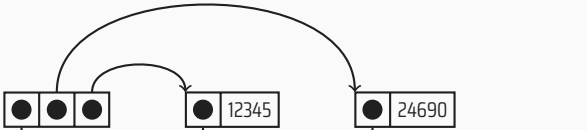

```
PLUS:
a=read thunkptr[1][1]
b=read thunkptr[2][1]
newptr=alloc 2
newptr[0]=DATA
newptr[1]=a+b;
thunkptr[0]=IND
thunkptr[1]=newptr
continue newptr
```

```
IND:
thunkptr=thunkptr[1]
jmp *thunkptr;
```

```
DATA:
continue thunkptr
```

STG objects are identified by the code that evaluates them

let $x = 12345$ **in** $x + x$




```
PLUS:
a=read thunkptr[1][1]
b=read thunkptr[2][1]
newptr=alloc 2
newptr[0]=DATA
newptr[1]=a+b;
thunkptr[0]=IND
thunkptr[1]=newptr
continue newptr
```

```
IND:
thunkptr=thunkptr[1]
jmp *thunkptr;
```

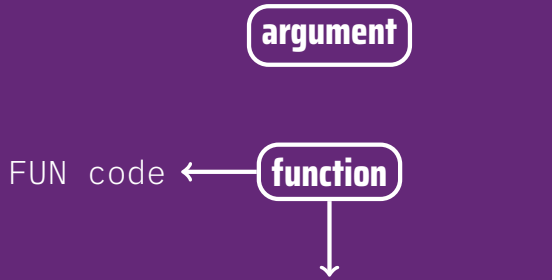
```
DATA:
continue thunkptr
```

STG evaluation proceeds by graph reduction

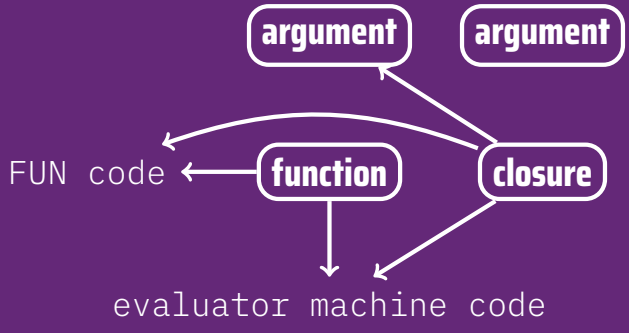
The main operation that STG programs do is called *forcing*. Roughly, forcing reads as:

> Hello object X, please evaluate yourself to a head-normal form
> and pass the result to this continuation.

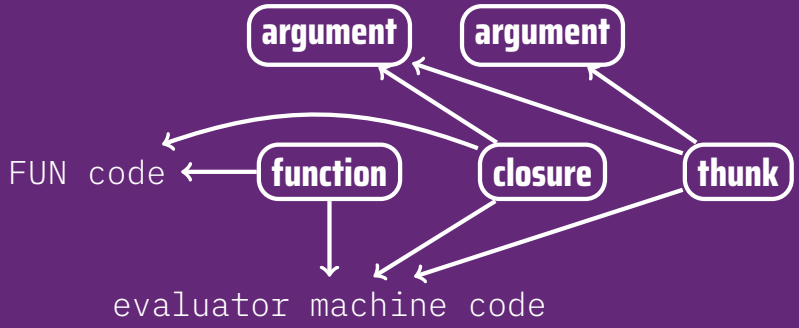
- Upon successful evaluation, the object (usually a thunk) rewrites itself with the result, possibly using an indirection. (This allows result sharing and memoization.)
- Evaluation continues by passing the result to continuation (directly).
- Continuations typically represent a “case” statement that can patternmatch on the head of the result.


argument
FUN code
function
evaluator machine code

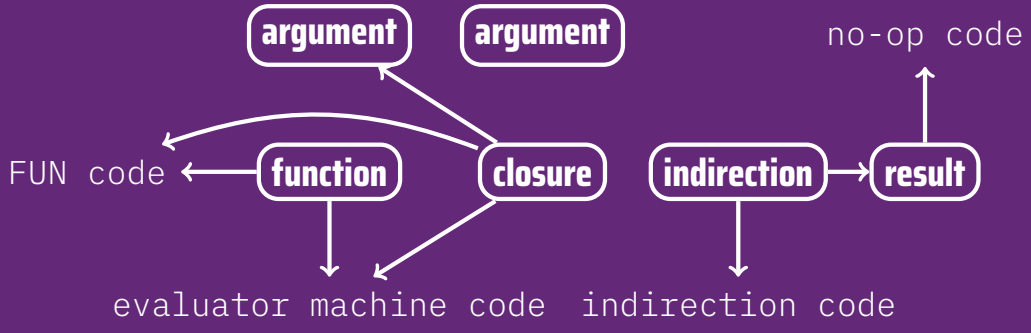
Life on STG


argument
argument
FUN code
function
closure
evaluator machine code

Life on STG


argument
argument
FUN code
function
closure
thunk
evaluator machine code

Life on STG


argument
argument
no-op code
FUN code
function
closure
indirection
result
evaluator machine code
indirection code

STG takeaway

In Haskell, each let-binding in code generates only one thunk upon each scope entry.

The first part of the program that forces the thunk replaces it by the result, which is re-used by other parts of the program.

STG takeaway

In Haskell, each let-binding in code generates only one thunk upon each scope entry.

The first part of the program that forces the thunk replaces it by the result, which is re-used by other parts of the program.

STG takeaway

In Haskell, each let-binding in code generates only one thunk upon each scope entry.

The first part of the program that forces the thunk replaces it by the result, which is re-used by other parts of the program.

STG takeaway

In Haskell, each let-binding in code generates only one thunk upon each scope entry.

The first part of the program that forces the thunk replaces it by the result, which is re-used by other parts of the program.

Outcomes:

- top-level binding with no parameters will be evaluated at most once
- this works: $fibs = 0 : 1 : zipWith\ (+)\ fibs\ (tail\ fibs)$
- monomorphism restriction

Low-level

- STG is converted to Cmm
  - Cmm is like C, but with external garbage collection support, and without any complex types or actual pointers
- Cmm can be compiled just like C to object code module `.o`
- The result is compiled with RTS, which adds:
  - OS-reachable `main`
  - allocator and GC
  - IO routines
  - implementation of lower levels of GHC.Prim

Low-level: modules

Modularization works just like with C++:

- `.hs` is like the source code in `.c` a `.cpp`
- `.hi` is like automatically generated header files for the sources
- `.o` is just like C-style `.o`, and can be transformed to `.so` and `.a` (or `.dll` and `.lib`) and linked as with C/C++

Historical note: Changes in `.hi` files (even breaking ones) used to creep into other `.hi`, and we used to have to nuke the whole `~/.cabal` and start from scratch. This problem was solved with nix-style builds in Cabal 2.

Curry-Howard correspondence

So what does a type of an expression actually mean?

- Traditional languages: format of the data (roughly)
- Functional programming: what is the “format” of $\longrightarrow$?

Motivational task

Write a function f, which has a type $f :: a \to b$.

Motivational task

Write a function f, which has a type $f :: a \to b$.

1. I get something of type a and discard it, OK.

Motivational task

Write a function f, which has a type $f :: a \to b$.

1. I get something of type a and discard it, OK.
2. I make up some b?

Motivational task

Write a function f, which has a type $f :: a \to b$.

1. I get something of type a and discard it, OK.
2. I make up some b?
   $(\lambda x \to 1) :: a \to \mathsf{Integer} \not\equiv a \to b$

$f :: a \to b$ means $f :: \forall a.\forall b.a \to b$. The program effectively promises that it will be able to create any b.

Motivational task

Write a function f, which has a type $f :: a \to b$.

1. I get something of type a and discard it, OK.
2. I make up some b?
   $(\lambda x \to 1) :: a \to \mathsf{Integer} \not\equiv a \to b$

$f :: a \to b$ means $f :: \forall a.\forall b.a \to b$. The program effectively promises that it will be able to create any b.

Haskell is conveniently broken so we can solve it as follows:

$$(\lambda \_ \to \bot) :: p \to a$$

Semantics of $\longrightarrow$

By using $a \longrightarrow b$ we make a proposition that we can convert a thing of type a to a thing of type b.

Semantics of $\longrightarrow$

By using $a \longrightarrow b$ we make a proposition that we can convert a thing of type a to a thing of type b.

Types as propositions (aka logic as existence of typed things)

- $\rightarrow$ can be reimagined as a logical implication
- instead of conjunctions $a \wedge b$ we will use tuples (a, b)
- instead of disjunctions $a \vee b$ we use $\mathsf{Either}\ a\ b$

For example, this obviously doesn't hold:

$$a \rightarrow (a, b) \qquad a \implies (a \wedge b)$$

...but if someone supplies a way to convert a's to b's, we can make it work:

$$(a \rightarrow b) \rightarrow (a \rightarrow (a, b)) \qquad (a \implies b) \implies (a \implies (a \wedge b))$$

Programs are proofs of their types

A program with type *t* exists if and only if a proposition that corresponds to *t* is provable in constructive propositional logic.

Programs as proofs

$$(\lambda x \to (x, \bot)) :: a \to (a, b)$$

(program contains an error (Haskell magic), the proposition is not proven)

$$(\lambda f\ x \to (x, f\ x)) :: (a \to b) \to (a \to (a, b))$$

(program exists in λ-calculus, the proposition is proven)

Correspondence

Axiom of logic	λ-calculus
Weakening ($\phi \implies (\psi \implies \phi)$)	$\vdash K : \alpha \to \beta \to \alpha$
Substitution	$S = (\lambda xyz.xz(yz))$
	$\vdash S : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$
Modus ponens	$\{A : \alpha \to \beta, B : \alpha\} \vdash AB : \beta$
	$\vdash I : (\alpha \to \beta) \to \alpha \to \beta$
Proof search	Type inhabitation

Extra correspondence:

- Combining basic axioms can build any proposition in constructive (intuitionistic) propositional logic.
- Any program can be built out of the SKI combinators
  (including the universal Turing machine)

Correspondence

Axiom of logic	λ-calculus
Weakening ($\phi \implies (\psi \implies \phi)$)	$\vdash K : \alpha \to \beta \to \alpha$
Substitution	$S = (\lambda xyz.xz(yz))$
	$\vdash S : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$
Modus ponens	$\{A : \alpha \to \beta, B : \alpha\} \vdash AB : \beta$
	$\vdash I : (\alpha \to \beta) \to \alpha \to \beta$
Proof search	Type inhabitation

Extra correspondence:

- Combining basic axioms can build any proposition in constructive (intuitionistic) propositional logic.
- Any program can be built out of the SKI combinators
  (including the universal Turing machine)

Example

I want to prove: $a \wedge (b \vee c) \implies (a \wedge b) \vee (a \wedge c)$

Example

I want to prove: $a \wedge (b \vee c) \implies (a \wedge b) \vee (a \wedge c)$

$$p :: (a, \text{Either } b\ c) \to \text{Either } (a, c)\ (a, b)$$

Example

I want to prove: $a \wedge (b \vee c) \implies (a \wedge b) \vee (a \wedge c)$

$$p :: (a, \text{Either } b\ c) \to \text{Either } (a, c)\ (a, b)$$

$$p\ (a, e) = \textbf{case}\ e\ \textbf{of}\ \text{Left}\ b \to \text{Right}\ (a, b)$$
$$\text{Right}\ c \to \text{Left}\ (a, c)$$

Benefit: typed term search is algorithmically much more easy&deterministic than search through logical formulas.

Main implementations: Agda, Coq, HOL, Djinn

What do we do with negation?

Logic	λ-calculus
True	$()$
False	Void
$\neg a$	$a \to \mathsf{Void}$
Theory contradicts itself	$\bot$

Void cannot be created, we can only 'forward' it if we get one.

Note: intuitionistic logic does not recognize that $\neg\neg a = a$!

What do we do with negation?

Logic	λ-calculus
True	$()$
False	Void
$\neg a$	$a \to \text{Void}$
Theory contradicts itself	$\bot$

Void cannot be created, we can only 'forward' it if we get one.

Note: intuitionistic logic does not recognize that $\neg\neg a = a$!

In Haskell view, *catch* is a double negation:

$$throw :: a \to \text{Void}$$
$$catch :: ((a \to \text{Void}) \to \text{Void}) \to a$$

What should I take home from this?

- Function is a proof of existence of data transformation
- Function code can be determined from its type
  - more precise types $\Longrightarrow$ less valid implementations
    $\Longrightarrow$ less errors
  - given a sufficiently precise type, implementation can be done by the computer quite reliably (try Coq&Agda)
  - you may need dependent types if you want to “request” specific behavior with values
  - generic types are usefully restrictive by not giving a possibly misleading advice
- Handling data specifications like this is quite useful
  (data have a type $\Longleftrightarrow$ data are valid)

What should I take home from this?

- Function is a proof of existence of data transformation
- Function code can be determined from its type
  - more precise types $\Longrightarrow$ less valid implementations
    $\Longrightarrow$ less errors
  - given a sufficiently precise type, implementation can be done by the computer quite reliably (try Coq&Agda)
  - you may need dependent types if you want to “request” specific behavior with values
  - generic types are usefully restrictive by not giving a possibly misleading advice
- Handling data specifications like this is quite useful
  (data have a type $\Longleftrightarrow$ data are valid)

Tight types

Code robustness increases with:

1. increasingly generic functions
2. increasingly specific data types

Generic functions work with more kinds of data, and their type restricts the admissible constructions, removing many invalid ones.

Precisely specified data simplify the program by removing the imprecision handlers (`if`s).

Not-really-tight types

Common imprecise data constructions:

- *null*-member instead of multiple data variants
- lists instead of exact *n*-tuples or records
- any variant of XML

Types too tight?

There's a compiler called CakeML which has (precise) types for all $\geqslant 20$ different intermediate code representations.

- Benefit: you can verify its correctness in 10 minutes
- New problem: you need to write lots of similar functions for $\geqslant 20$ data types
- Solution:

Types too tight?

There's a compiler called CakeML which has (precise) types for all ⩾20 different intermediate code representations.

- Benefit: you can verify its correctness in 10 minutes
- New problem: you need to write lots of similar functions for ⩾20 data types
- Solution: feature-based type classes that give generic access to common data operations:
  - For example: `HasVars`, `Formattable`, …
  - THIH:
    - `Type`, `Pred`, `Qual` t, `Scheme`, `Assump` ∈ `Types`
    - `Type`, [a], `Pred`, `Qual` t ∈ `Instantiate`
  - `HasField` for data types with fields of conflicting names.

Extra: Assumptions are tangible objects!

Types can help us to track assumptions:

- Num a gives us a safe assumption we can add as together
- $a \to b$ gives us a possibility to produce bs from as

There are also many other kinds of assumptions flying around:

- **if** $x > 5$ gives us:
  - assumption that x is at least 6 in the if-branch
  - assumption that x is at most 5 in the else-branch
- pattern matching connects the assumption production ("it is $(x : xs)$!") to the availability of outcomes (the bound variables)
- with dependent types we could track these too:

$$length :: (\text{Num } i, i \geqslant 0) \Rightarrow [a] \to i$$

Use assumptions for writing better code

Check where your assumption comes from.

Check you used all assumptions you got.

Extra: Parametricity

Parametricity is the property of pure languages:

- behavior of functions is completely determined by their parameters
- behavior of polymorphic functions is determined by their type parameters

✎Theorems for free! Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA '89), ACM. (Philip Wadler, 1989)

Parametricity gives assumptions directly from types

Without ever looking at actual definitions of any functions, you can mechanically derive that:

- For any function $f :: [a] \to [b]$ and total g, this composition is safe:
  $map\ g \cdot f \equiv f \cdot map\ g$
- Function $sort :: (a \to a \to \mathsf{Bool}) \to [a] \to [a]$ composes:
  f is monotonic $\implies sort\ p \cdot map\ f \equiv map\ f \cdot sort\ p$
- All functions of type $(a \to b) \to [a] \to [b]$ are less-general variations on map.
- A truly generic equality function $eq :: a \to a \to \mathsf{Bool}$ can not be defined.
- For any a, b where $b\ (x + y) = a\ x + b\ y$ and $b\ z = z'$:
  $b \cdot foldr\ (+)\ u \equiv foldr\ (+)\ u' \cdot map\ a$

Takeaways from parametricity

Generic functions can't do specific operations.

That gives them less ways to mess it up.