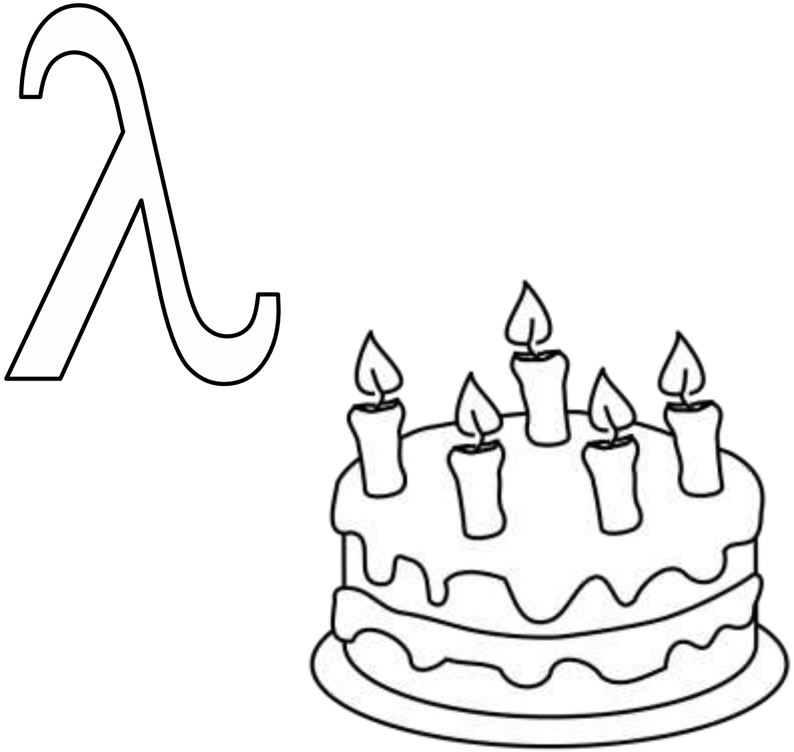
The Best of Both Worlds

J. Garrett Morris

Garrett.Morris@ed.ac.uk

Two of my favorite things



By their powers combined

$$\lambda x. \lambda y. x$$

- Types of arguments: x can be instantiated freely, y can only be unrestricted
- Type of $\lambda y. V$ depends on type of V: if V is linear, function must be as well

By their powers combined

$$\lambda x.\lambda y.x : \begin{cases} t \to^{\bullet} u \to^{\circ} t & \text{if } u \text{ un} \\ t \to^{\bullet} u \to^{\bullet} t & \text{if } t, u \text{ un} \end{cases}$$

- Types distinguish whether functions can be copied or discarded
- Central problem: generic combinator programming with multiple function types

The quill is mightier…

We introduce a Qualified Linear Language

- Integrates linear and polymorphic functional programming, using predicates on types
- Principal types (and decidable type inference)
- Conservative extension of existing functional (term) languages.

Builds on recent work on functional linear programming:

- Mazurak et al., "Lightweight linear types in System F°"
- Tov & Pucella, "Practical Affine Types"

Linearity and overloading

$$\lambda x. x + x$$

- Type of x must be numeric and unrestricted.
- Characterize unrestricted-ness using same tools as characterize numeric-ness.

Linearity with class

$$\lambda x.\, \mathrm{let}\ (y, z) = \mathrm{dup}\ x\ \mathrm{in}\ y + z$$

- Unrestricted types have methods:

$$\begin{aligned} \mathrm{dup} &:: t \to t \otimes t \\ \mathrm{drop} &:: t \ \to 1 \end{aligned}$$

- Corresponds to interpretation of exponential modality via a commutative comonoid (Filinski, Seely)

Linearity with class

$$\lambda x.\,\mathrm{let}\,(y,z) = \mathrm{dup}\,x\ \mathrm{in}\ y + z :$$
$$(\mathrm{Num}\ t, \mathrm{Un}\ t) \Rightarrow t \to^{\bullet} t$$

- Unrestricted types have methods:

$$\mathrm{dup} :: t \to t \otimes t$$
$$\mathrm{drop} :: t \to 1$$

- Corresponds to interpretation of exponential modality via a commutative comonoid (Filinski, Seely)

Products and sums

$$\begin{aligned}\lambda\,(x,y).\ &\mathsf{let}\ (x',x'') = \mathrm{dup}\ x\ \mathsf{in}\\ &\mathsf{let}\ (y',y'') = \mathrm{dup}\ y\ \mathsf{in}\\ &\big((x',y'),(x'',y'')\big)\end{aligned}$$

- Duplication of products depends on corresponding operations for components.
- Can be captured by “class instances”:
  $\mathsf{instance}\ (\mathrm{Un}\ t, \mathrm{Un}\ u) \Rightarrow \mathrm{Un}\ (t \otimes u)\ \mathsf{where}\ \ldots$
  $\mathsf{instance}\ (\mathrm{Un}\ t, \mathrm{Un}\ u) \Rightarrow \mathrm{Un}\ (t \oplus u)\ \mathsf{where}\ \ldots$

Things best left unstated

$$\lambda x.\, x + x : (\mathrm{Num}\; t, \mathrm{Un}\; t) \Rightarrow t \to^{\bullet} t$$

- Introduction of dup and drop implied by reuse or disuse of variables.

An application

$$\lambda\,(f, x).\,f\ x$$

- Safe for both linear and unrestricted functions; want to avoid repetition of combinators
- Syntax of application overloaded to apply to both varieties of functions
- Reflect using qualified types (but not a type class)

An application

$$\lambda\,(f, x).\, f\ x : \mathrm{Fun}\ f \Rightarrow f\ t\ u \otimes t \to^{\bullet} u$$

- Fun class ranges over function types ($\to^{\circ}$ and $\to^{\bullet}$).
- Syntactic sugar:

$$t \to^{f} u \equiv \mathrm{Fun}\ f \Rightarrow f\ t\ u$$
$$t \to u \equiv t \to^{f} u\ (f\ \mathrm{fresh})$$

Another application

$$\lambda f.\lambda x.f\ x$$

- Linearity of partial application $\lambda x.V\ x$ depends on type of V.

Another application

$$\lambda f.\lambda x.f\ x : \begin{cases} (t \to^{\circ} u) \to t \to^{\circ} u \\ (t \to^{\bullet} u) \to t \to^{\circ} u \\ (t \to^{\bullet} u) \to t \to^{\bullet} u \end{cases}$$

- Key point: overloading of λ for constructing functions.
- Relationship: function on the left must be “more unrestricted”

Another application

$$\lambda f.\lambda x.f\ x : f \geq g \Rightarrow \left(t \rightarrow^{f} u\right) \rightarrow t \rightarrow^{g} u$$

- $\tau \geq \upsilon$ means τ has as many structural rules as υ
- E.g., $(\rightarrow^{\bullet}) \geq (\rightarrow^{\circ})$

A constant example

$$\lambda x.\lambda y.x : (t \geq f, \mathrm{Un}\, u) \Rightarrow t \to u \to^{f} t$$

- Use of $\tau \geq \upsilon$ predicates isn’t limited to functions.

Consider the functor

$$\text{class Functor } h \text{ where}$$
$$\quad \text{fmap} :: (t \to u) \to (h\, t \to h\, u)$$

- Prototypical Haskell-like abstraction pattern
- Question: what do to about the arrows
- Example in the paper: monads

Some maps are more equal

$$\begin{aligned} \mathrm{fmap1}\ f\ [\,] &= [\,] \\ \mathrm{fmap1}\ f\ (x : xs) &= f\ x : \mathrm{fmap1}\ f\ xs \end{aligned}$$

- Based on the Haskell functor instance for lists
- Lifted function f duplicated in the “cons” case
- So, f must have type $t \to^{\bullet} u$

Some maps are more equal…

$$\begin{aligned}\mathrm{fmap2}\ f\ sf &= \\ &\lambda s.\,\mathrm{let}\ (z, s') = sf\ s\ \mathrm{in}\ (f\ z, s')\end{aligned}$$

- Based on the Haskell functor instance for state transformers
- Lifted function f only needs to be unrestricted if the resulting state transformer is unrestricted

Generalizing over linearity

$$\text{class Functor}\ h\ f\ |\ h\ \rightarrow\ f\ \text{where}$$
$$\quad \text{fmap}\ ::\ (t\ \rightarrow^{f}\ u) \rightarrow^{\bullet} (h\,t\ \rightarrow^{f}\ h\,u)$$

- Functor type determines the type of its maps
- No “more” polymorphism than in Haskell

Generalizing over linearity

$\text{instance Functor } [\,]\ (\rightarrow^{\bullet})\ \text{where } \ldots$

$\text{type State } k\ s\ t = s \rightarrow^{k} (t \otimes s)$
$\text{instance Functor (State } k\ s)\ k\ \text{where } \ldots$

- Functor type determines the type of its maps
- No “more” polymorphism than in Haskell

But wait, there's more…

$$\text{data } T_1 \text{ where } MkT_1 \;::\; a \rightarrow^{\bullet} T_1$$
$$\text{data } T_2 \text{ where } MkT_2 \;::\; \text{Un}\, a \Rightarrow a \;\rightarrow^{\bullet} T_2$$

- T_1 and T_2 differ only in their linearity.
- Same pattern as functions.

But wait, there's more…

$$\begin{aligned}
&\text{class } T\ t \text{ where}\\
&\quad MkT :: (a \geq t) \Rightarrow a \rightarrow^{\bullet} t\\
&\quad unT :: (f \geq g)\\
&\qquad\qquad \Rightarrow \left(\forall a.\, a \rightarrow^{f} b\right) \rightarrow t \rightarrow^{g} b
\end{aligned}$$

- MkT's use of $\geq$ similar to that in typing of λ
- Use of $\geq$ in unT from capturing case body as function

The shoulders of giants

Linear functional calculi

- Mazurak et al., “Lightweight linear types in System F°”
- Tov & Pucella, “Practical Affine Types”
- Gay & Vasconcelos, “Linear type theory for asynchronous session types”

Uniqueness and usage types

- Smetsers et al., “Guaranteeing safe destructive updates through a type system with uniqueness information for graphs”
- Gustavsson & Svenningsson. “A usage analysis with bounded usage polymorphism and subtyping”
- Hage et al. “A generic usage analysis with subeffect qualifiers”

Things you haven’t seen

Examples

- Session types
- Monads

Metatheory

- Principal types and type inference
- Type safety
- Conservative extension of existing functional languages

Prototype implementation…. coming very soon.