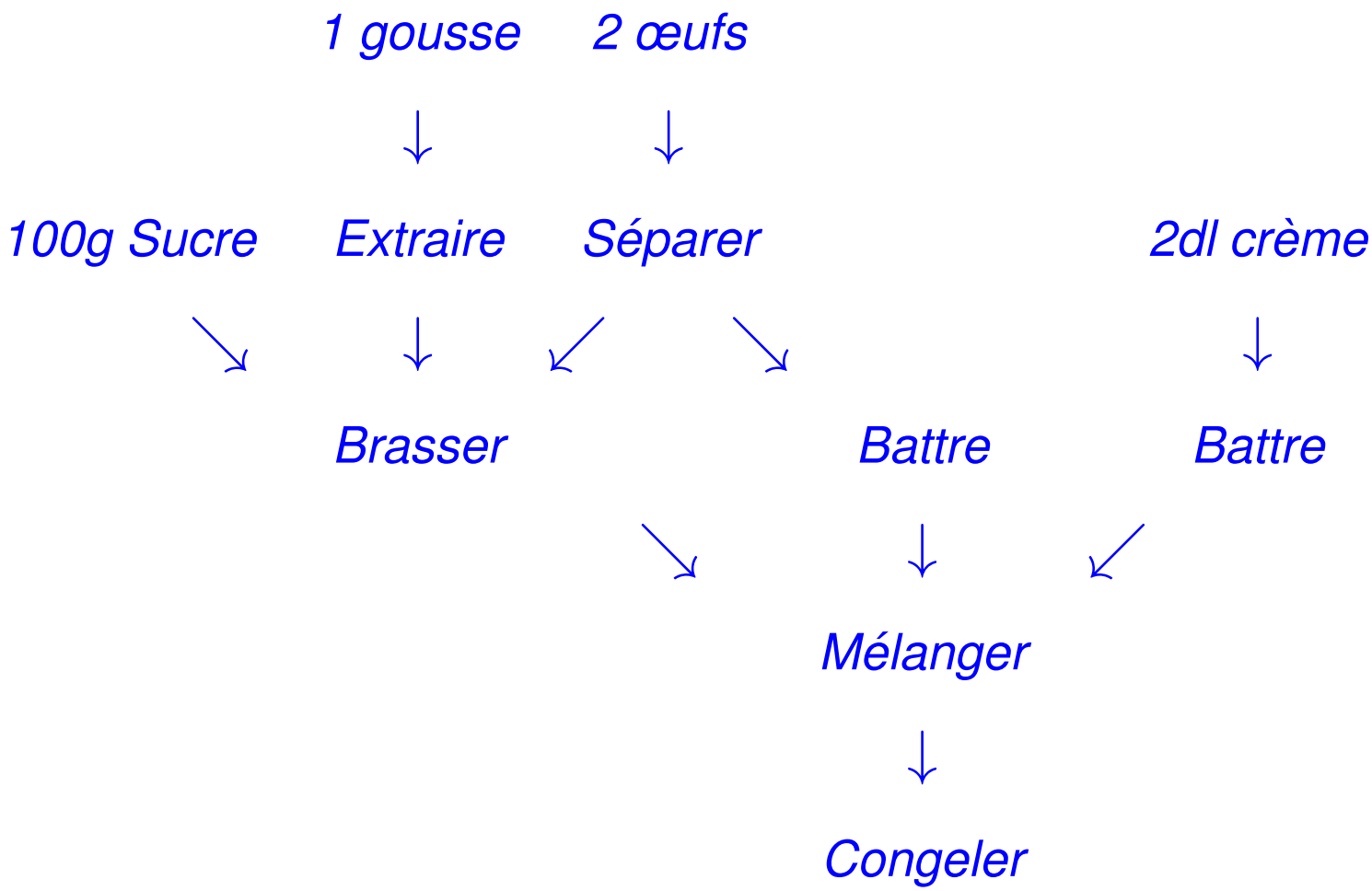
Glace vanille impérative

1. Séparer les jaunes et les blancs de 2 œufs
2. Garder les blancs au réfrigérateur!
3. Ajouter $\frac{1}{2}$ tasse de sucre aux jaunes!
4. Y ajouter les graines d'une (voire $1\frac{1}{2}$) gousse de vanille!
5. Brasser vigoureusement (ça devient un peu blanchâtre)!
6. Battre 2dl de crème!
7. Battre les blancs en neige!
8. Mélanger le tout et mettre au congélateur!

Glace vanille fonctionnelle



Lambda-calcul

Inventé en 1941 par Alonzo Church

La base de la théorie des langages de programmation

Inspiration de Lisp, Scheme, ML, Haskell, ...

$$e ::= c \mid x \mid \lambda x \to e \mid e_1\ e_2$$

Sémantique:

$$(\lambda x \to e_1)\ e_2 \quad \leadsto \quad e_1[e_2/x] \qquad \beta\textit{-réduction}$$

Il y a aussi le renommage-α et la réduction-η

Chapitre 1 de Hudak, 8 et 14 de Sethi, 5 de Pierce

Haskell

Langages de la famille: Haskell, OCaml, Reason, F#, Coq, SML

Utilisés dans l'industrie par:

Jane Street, Standard Chartered, J.P. Morgan, Facebook, BAE Systems, Crédit Suisse, Microsoft, Docker, Wolfram, ...

Sucre syntaxique en Haskell

Haskell permet la notation infixe et préfixe: $1 + 2$ ou $(+)\ 1\ 2$

À l'inverse: $\mathtt{mod}\ x\ y$ ou $x\ `\mathtt{mod}`\ y$

Définition de fonction:

$$\mathit{double}\ o\ x = o\ x\ x \quad \Leftrightarrow \quad \mathit{double} = \lambda\, o \to \lambda\, x \to o\ x\ x$$

Exemple $(*)\ `\mathit{double}`\ 7$:

$$(\lambda o \to \lambda x \to o\ x\ x)\ (*)\ 7 \rightsquigarrow (\lambda x \to (*)\ x\ x)\ 7 \rightsquigarrow (*)\ 7\ 7 \rightsquigarrow 4$$

Expressions et valeurs

Une *valeur* est une expression irréductible

I.e. “pas de β-redex”

Exemples: 2, 781, $\lambda x \to [x, x]$, ...

Évaluer = réduire une expression à une valeur

Est-ce que cela peut toujours se réduire à une valeur?

Récursion

$$\mathit{power}\ x\ 0 = 1$$

$$\mathit{power}\ x\ y = x * (\mathit{power}\ x\ (y - 1))$$

$$\begin{aligned}
&\mathit{power}\ 3\ 2 \rightsquigarrow 3 * (\mathit{power}\ 3\ (2 - 1)) \\
&\rightsquigarrow 3 * (\mathit{power}\ 3\ 1) \rightsquigarrow 3 * (3 * (\mathit{power}\ 3\ (1 - 1))) \\
&\rightsquigarrow 3 * (3 * (\mathit{power}\ 3\ 0)) \rightsquigarrow 3 * (3 * 1) \rightsquigarrow 3 * 3 \rightsquigarrow 9
\end{aligned}$$

Un autre exemple: $\mathit{double}\ \mathit{power}\ 3$

Ordre d'évaluation

¿ ¿ *double* $(*)$ $(1+2)$? ?

Ordre d'évaluation

¿ ¿ *double* $(*)$ $(1+2)$? ?

¡Indifférent!

$$\textit{double}\ (*)\ (1+2) \rightsquigarrow \textit{double}\ (*)\ 3 \rightsquigarrow 3*3 \rightsquigarrow 9$$

$$\textit{double}\ (*)\ (1+2) \rightsquigarrow (1+2)*(1+2) \rightsquigarrow \ldots \rightsquigarrow 9$$

Propriété fondamentale d'un langage fonctionnel pur

L'ordre importe quand même

$$\mathit{power}\ x\ y = \texttt{if}\ y = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (\mathit{power}\ x\ (y-1))$$

$\mathit{power}\ 3\ 0$

$\rightsquigarrow\ \texttt{if}\ 0 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3 * (\mathit{power}\ 3\ (0-1))$

$\rightsquigarrow\ \texttt{if}\ 0 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3 * (\mathit{power}\ 3\ (-1))$

$\rightsquigarrow\ \texttt{if}\ 0 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3 * (\texttt{if}\ -1 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3 * (p$

Haskell garanti qu'il termine si c'est possible

Transparence référentielle

e_1 et e_2 sont strictement équivalentes si $e_1 \rightsquigarrow^* e_3$ et $e_2 \rightsquigarrow^* e_3$

Remplacer e_1 par e_2 dans une expression e ne change pas le résultat

Modulo renommage α, bien sûr

C'est pour cela que l'ordre d'évaluation n'importe pas

Types

Système de classification qui a deux origines indépendentes:

- Logique mathématique: introduits pour éviter des problèmes tels que le paradoxe de Russell
- Langages de programmation: besoin de distinguer des valeurs de natures différentes (e.g. différente taille)

Un *type* est comme un ensemble d'objets similaires

Similaires = acceptent les même opérations

Genres de typages

Un langage peut utiliser les types de deux manières:

- Typage dynamique: les *valeurs* portent leur type

  N'importe quelle variable peut contenir n'importe quelle valeur

- Typage statique: le type est associé aux *variables*

  Une variable ne peut contenir que des valeurs du type spécifié

Le typage est dit *fort* ou *faible* selon s'il est possible d'utiliser une opération sur une valeur du mauvais type

Certains langages ne sont simplement pas typés du tout!

Types en Haskell

¿ ¿ $3 + \mathit{power}$? ?

¿ ¿ $\mathit{double}\ 1\ 2$? ?

Types en Haskell

¿ ¿ $3 + \mathit{power}$? ?

¿ ¿ $\mathit{double}\ 1\ 2$? ?

Chaque expression e a un type τ:

Notation Haskell: $e :: \tau$

Si $v_1 :: \tau$ et $v_2 :: \tau$, ils peuvent être utilisés aux mêmes endroits

$3 :: \texttt{Int}$, $3.14 :: \texttt{Float}$, $\texttt{True} :: \texttt{Bool}$,

power :: $\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}$

Polymorphisme

Certaines fonctions peuvent avoir plusieurs types:

$$\textit{id\_i} :: \texttt{Int} \to \texttt{Int}$$

$$\textit{id\_i}\ x = x$$

$$\textit{id\_s} :: \texttt{String} \to \texttt{String}$$

$$\textit{id\_s}\ x = x$$

On peut alors utiliser une *variable de type*:

$$\textit{id} :: \alpha \to \alpha$$

$$\textit{id}\ x = x$$

On dit alors que la fonction est *polymorphe*

Inférence de types

En Haskell, il n'est pas indispensable d'écrire les types

```
% hugs
[...]
Prelude> :type \ o x -> o x x
\o x -> o x x :: (a -> a -> b) -> a -> b
Prelude>
```

Les annotations de type sont recommandées:

- Meilleurs messages d'erreur
- Documentation

Sémantique statique de STLC

$$\Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \to e : \tau_1 \to \tau_2}$$

$$\Gamma \vdash n : \texttt{Int} \qquad \Gamma \vdash (+) : \texttt{Int} \to \texttt{Int} \to \texttt{Int}$$

Types, méthodes formelles, vérification

Vérification $\neq$ Tests

Méthodes formelles:

- Logique de Hoare: lourd, coûteux, Sisyphe
- Vérification de modèle: plus léger, plus limité, pas toujours formel
- Génération automatique de code: bugs dans la spec?
- Types: très légers, très limités, très formels

Seuls les types sont utilisés à grande échelle

Hardware en avance sur le software

Types structurés

Paires: (e_1, e_2), `fst`, `snd` de type: (τ_1, τ_2)

Listes: $[]$, $e_h : e_t$, $[e_1, ..., e_n]$, `head`, `tail` de type: $[\tau]$

$$(1, 2) : (\texttt{Int}, \texttt{Int})$$

$$(1, (2, 3)) : (\texttt{Int}, (\texttt{Int}, \texttt{Int}))$$

$$(1, [2, 3]) : (\texttt{Int}, [\texttt{Int}])$$

$$[1, [2, 3]] : \textit{Error}$$

$$[[1], [2, 3]] : [[\texttt{Int}]]$$

$$[(1, 2), (2, 3)] : [(\texttt{Int}, \texttt{Int})]$$

Exemples

$$\mathit{length} :: [\alpha] \to \texttt{Int}$$

$$\mathit{length}\ [] = 0$$

$$\mathit{length}\ (\_ : xs) = \mathit{length}\ xs + 1$$

Exemples

$$\mathit{length} :: [\alpha] \to \texttt{Int}$$

$$\mathit{length}\ [] = 0$$

$$\mathit{length}\ (\_ : xs) = \mathit{length}\ xs + 1$$

$$\mathit{zip} :: ([\alpha], [\beta]) \to [(\alpha, \beta)]$$

$$\mathit{zip}\ ([], \_) = []$$

$$\mathit{zip}\ (\_, []) = []$$

$$\mathit{zip}\ (x : xs, y : ys) = (x, y) : \mathit{zip}\ (xs, ys)$$

Exemples

$$\mathit{length} :: [\alpha] \to \texttt{Int}$$

$$\mathit{length}\ [] = 0$$

$$\mathit{length}\ (\_ : xs) = \mathit{length}\ xs + 1$$

$$\mathit{zip} :: ([\alpha], [\beta]) \to [(\alpha, \beta)]$$

$$\mathit{zip}\ ([], \_) = []$$

$$\mathit{zip}\ (\_, []) = []$$

$$\mathit{zip}\ (x : xs, y : ys) = (x, y) : \mathit{zip}\ (xs, ys)$$

Pourquoi pas:

$$\mathit{length}\ xs = \texttt{if}\ xs == []\ \texttt{then}\ 0\ \texttt{else}\ \mathit{length}\ (\mathit{tail}\ xs) + 1$$

Polymorphisme avec Type Classes

$$\begin{aligned} &\mathit{applymin}\ f\ x\ y \\ &\quad \mid x \leq y = f\ x\ y \\ &\quad \mid \texttt{otherwise} = f\ y\ x \end{aligned}$$

x et y peuvent être de type $\texttt{Float}$, $\texttt{Int}$, mais pas $\texttt{Int} \to \texttt{Int}$

$$\mathit{applymin} :: \texttt{Ord}\ \alpha \Rightarrow (\alpha \to \alpha \to \beta) \to \alpha \to \alpha \to \beta$$

Même chose pour l'opération d'égalité, l'addition, ...:

$$(==) :: \texttt{Eq}\ \alpha \Rightarrow \alpha \to \alpha \to \texttt{Bool}$$

$$(+) :: \texttt{Num}\ \alpha \Rightarrow \alpha \to \alpha \to \alpha$$

Filtrage

- Variable: accepte tout et le lie à la variable
- Filtre spécial _: accepte tout, ne lie rien
- Constante: accepte seulement une value de la bonne forme et seulement si les sous-éléments sont acceptés par les sous-filtres
- Garde: expression booléenne quelconque
- Filtre-OU: $(f_1|f_2|f_3)$ accepte une valeur ssi elle est acceptée par f_1, f_2, ou f_3 et lie les mêmes variables
- Filtre-AS: $(x \texttt{ as } f)$ comme f mais en plus lie la valeur à x
- Filtres répétés: par exemple (x, x)

Exemple de filtre

$$\begin{aligned}
&\mathit{merge} :: \texttt{Ord}\,\alpha \Rightarrow ([\alpha],[\alpha]) \to [\alpha] \\
&\mathit{merge}\,((xs,[])|([],xs)) \;= xs \\
&\mathit{merge}\,(a\texttt{@}(x:xs), b\texttt{@}(y:ys)) \\
&\qquad | \; x \leq y = x : \mathit{merge}\,(xs,b) \\
&\qquad | \; \texttt{otherwise} = y : \mathit{merge}\,(a,ys)
\end{aligned}$$

Exemple de filtre, le retour

$$\mathit{merge} :: \mathtt{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha]$$

$$\mathit{merge}\ xs\ []\ = xs$$

$$\mathit{merge}\ []\ xs\ = xs$$

$$\mathit{merge}\ (x : xs)\ (b\ @\ (y : \_)) \mid x \leq y = x : \mathit{merge}\ xs\ b$$

$$\mathit{merge}\ a\ (y : ys) = y : \mathit{merge}\ a\ ys$$

Polymorphisme en λ-calcul

Le λ-calcul typé avec polymorphisme s'appelle *System F*

Cœur des langages OCaml/Haskell (en théorie et en pratique)

$$e ::= c \mid x \mid \lambda x : \tau \to e \mid e_1\, e_2 \mid \Lambda t \to e \mid e[\tau]$$

$$\tau ::= \texttt{Int} \mid \tau_1 \to \tau_2 \mid t \mid \forall t.\tau$$

La fonction identité est en fait traduite comme suit:

$$\mathit{id} :: \forall \alpha.\alpha \to \alpha$$

$$\mathit{id} = \Lambda\alpha \to \lambda x : \alpha \to x$$

$$\mathit{réponse} = \mathit{id}[\texttt{Int}]\, 42$$

Inventé en 1972/1974 par Girard/Reynolds

Sémantique statique de System F

$$\Gamma, x : \tau \vdash x : \tau$$

$$\frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \to e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda t \to e : \forall t.\tau} \qquad \frac{\Gamma \vdash e : \forall t.\tau_1}{\Gamma \vdash e[\tau_2] : \tau_1[\tau_2/t]}$$

Curry-Howard

Intime connection entre logique et langages de programmation

$$\frac{\Gamma \vdash e_1 : \alpha \to \beta \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1\ e_2 : \beta} \quad \text{vs} \quad \frac{P_1 \Rightarrow P_2 \qquad P_1}{P_2}$$

La règle de typage de l'application correspond au *modus ponens*

La β-réduction correspond au *cut-elimination*

Type = Proposition; Programme = Preuve

$$\frac{\Gamma \vdash e : \forall t.\tau_2}{\Gamma \vdash e[\tau_1] : \tau_2[\tau_1/t]} \quad \text{vs} \quad \frac{\forall x.P}{P[e/x]}$$

Il n'existe pas de fonction de type $\forall t_1, t_2.t_1 \to t_2$

Créer de nouveaux types

Types de données algébriques

$$\begin{aligned}\texttt{data}\ t = \ & \mathit{Tag}_1\ \tau_{11}\ \ldots\ \tau_{1n} \\ \mid\ & \mathit{Tag}_2\ \tau_{21}\ \ldots\ \tau_{2n'} \\ \mid\ & \ldots \\ \mid\ & \mathit{Tag}_m\ \tau_{m1}\ \ldots\ \tau_{mn''}\end{aligned}$$

Listes: $\texttt{data}\ \mathit{List}\ \alpha\ =\ \mathit{Nil}\ \mid\ \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$

Produits: $\texttt{data}\ \mathit{Pair}\ \alpha\ \beta\ =\ \mathit{Pair}\ \alpha\ \beta$

Sommes: $\texttt{data}\ \mathit{Either}\ \alpha\ \beta\ =\ \mathit{Left}\ \alpha\ \mid\ \mathit{Right}\ \beta$

On peut aussi créer des alias avec $\texttt{type}\ t = \tau$

FIXME Algebraic data types

FIXME More about data types!

E.g. how they relate to subtyping

How they relate to structs/object/records/tuples

FIXME Type equality

FIXME structural vs names

Exemples sur arbres

$$\texttt{data}\ \mathit{Tree}\ \alpha = \mathit{Leaf} \mid \mathit{Node}\ (\mathit{Tree}\ \alpha)\ \alpha\ (\mathit{Tree}\ \alpha)$$

$$\mathit{sum} :: \mathit{Tree}\ \texttt{Int} \to \texttt{Int}$$
$$\mathit{sum}\ \mathit{Leaf} = 0$$
$$\mathit{sum}\ (\mathit{Node}\ t_l\ n\ t_r) = \mathit{sum}\ t_l + n + \mathit{sum}\ t_r$$

$$\mathit{insert} :: \mathit{Tree}\ \texttt{Int} \to \texttt{Int} \to \mathit{Tree}\ \texttt{Int}$$
$$\mathit{insert}\ \mathit{Leaf}\ n = \mathit{Node}\ \mathit{Leaf}\ n\ \mathit{Leaf}$$
$$\mathit{insert}\ (\mathit{Node}\ t_l\ m\ t_r)\ n$$
$$\quad \mid n < m = \mathit{Node}\ (\mathit{insert}\ t_l\ n)\ m\ t_r$$
$$\quad \mid \texttt{otherwise} = \mathit{Node}\ t_l\ m\ (\mathit{insert}\ t_r\ n)$$

Manipulation de listes

$$(++) :: [a] \to [a] \to [a]$$

$$[] ++ ys = ys$$

$$(x : xs) ++ ys = x : (xs ++ ys)$$

Est-ce que $xs ++ [] \equiv xs$?

Est-ce qu'on peut le prouver ?

Qu'est-ce que cela implique ?

Efficacité de la concaténation

Combien d'opérations sont-elle nécessaires pour évaluer:

$$([1,2,3] ++ [4,5,6]) ++ [7,8,9]$$

Qu'en est-il de

$$[1,2,3] ++ ([4,5,6] ++ [7,8,9])$$

Qu'elle est l'associativité de $++$?

Inverser les listes

$$\mathit{reverse} :: [a] \to [a]$$

$$\mathit{reverse}\ [] = []$$

$$\mathit{reverse}\ (x : xs) = ??$$

Répétition uniforme

$$\texttt{type}\ \mathit{Polygon} = [(\texttt{Int}, \texttt{Int})]$$

$$\mathit{transX} :: \texttt{Int} \to \mathit{Polygon} \to \mathit{Polygon}$$

$$\mathit{transX}\ o\ [] = []$$

$$\mathit{transX}\ o\ ((x, y) : vs) = (o + x, y) : \mathit{transX}\ o\ vs$$

$$\mathit{toupper}\ [] = []$$

$$\mathit{toupper}\ (c : cs) = \texttt{let}\ c' = \texttt{if}\ c \geq \text{'}a\text{'}\ \&\&\ c \leq \text{'}z\text{'}\ \texttt{then}\ c - \text{'}a\text{'} + \text{'}A\text{'}\ \texttt{else}\ c\ \texttt{in}\ c' : \mathit{toupper}\ cs$$

Fonctions d'ordre supérieur

Les fonctions sont des objets normaux: objets de *première classe*

On peut les passer en argument

Les renvoyer comme valeur de retour

Les stocker dans des structures de données

Currying

$$\mathit{double}\ o\ x = o\ x\ x$$

En réalité:

$$\mathit{double} \equiv \lambda o \to \lambda x \to o\ x\ x$$

Donc

$$\mathit{double}\ (+) \rightsquigarrow \lambda x \to (+)\ x\ x$$

Map

$$\mathit{map} :: (a \to b) \to ([a] \to [b])$$

$$\mathit{map}\ f\ [] = []$$

$$\mathit{map}\ f\ (x : xs) = f\ x : \mathit{map}\ f\ xs$$

$$\mathit{transX}\ o\ vs = \mathit{map}\ (\lambda(x, y) \to (o + x, y))\ vs$$

$$\mathit{toupper}\ s = \mathit{map}\ \mathit{toupper'}\ s$$

$$\texttt{where}\ \mathit{toupper'}\ c = \texttt{if}\ c \geq \text{'}a\text{'}\ \&\&\ c \leq \text{'}z\text{'}\ \texttt{then}\ c - \text{'}a\text{'} + \text{'}A\text{'}\ \texttt{else}\ c$$

Réduction η

En plus de la réduction β et de l'équivalence α, le λ-calcul défini aussi la réduction η:

$$\lambda x \to e\ x \quad \rightsquigarrow \quad e$$

Cette règle n'est pas utilisée aussi couramment

Variantes:

$$(\texttt{fst}\ e, \texttt{snd}\ e) \quad \rightsquigarrow \quad e$$

$$\texttt{if}\ e\ \texttt{then}\ \textit{True}\ \texttt{else}\ \textit{False} \quad \rightsquigarrow \quad e$$

Réductions de listes

$$sum :: [\texttt{Int}] \to \texttt{Int}$$

$$sum\ [] = 0$$

$$sum\ (x : xs) = x + sum\ xs$$

$$prod :: [\texttt{Int}] \to \texttt{Int}$$

$$prod\ [] = 1$$

$$prod\ (x : xs) = x * prod\ xs$$

Un réducteur générique

$$\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b$$

$$\mathit{foldr}\ op\ i\ [] = i$$

$$\mathit{foldr}\ op\ i\ (x : xs) = x\ `op`\ \mathit{foldr}\ op\ i\ xs$$

$$\mathit{sum} = \mathit{foldr}\ (+)\ 0$$

$$\mathit{prod} = \mathit{foldr}\ (*)\ 1$$

$$\mathit{concat} = \mathit{foldr}\ (++)\ []$$

Remarque (η-réduction sur les listes): $\mathit{foldr}\ (:)\ []\ xs \equiv xs$

Efficacité

$$\mathit{foldl} :: (a \to b \to a) \to a \to [b] \to a$$

$$\mathit{foldl}\ op\ i\ [] = i$$

$$\mathit{foldl}\ op\ i\ (x : xs) = \mathit{foldl}\ op\ (i\ `op`\ x)\ xs$$

$$\mathit{flip}\ f\ x\ y = f\ y\ x$$

$$\mathit{revcons} = \mathit{flip}\ (:)$$

$$\mathit{reverse} = \mathit{foldl}\ \mathit{revcons}\ []$$

Noms et portée

Un même *identificateur* peut désigner plusieurs choses

Une *déclaration* donne un sens à un identificateur

La *portée* d'une déclaration = le région du programme où l'identificateur réfère à cette déclaration

À l'inverse, les règles de portée définissent pour chaque usage d'une variable, à quelle déclaration il se réfère

Chap. 5.3 de Sethi

Portée lexicale

La *portée* et dite *statique* ou *lexicale* si elle est délimitée textuellement

La déclaration correspondant à un usage de variable est la déclaration précédente la plus *proche* dans le texte

Plusieurs déclarations d'un même identificateur peuvent être actives

$$\begin{aligned}
&\texttt{let}\ \mathit{double}\ o\ x = o\ x\ x \\
&\qquad f = \mathit{double}\ (+) \\
&\qquad g = \mathit{double}\ (*) \\
&\qquad o = \lambda o \to o \\
&\texttt{in}\ f\ 3 + g\ 4
\end{aligned}$$

Les droits fondamentaux des variables

Dans une expression e on dit qu'une variable x est *libre* si elle est utilisée sans être définie par e.

Expression	Variables libres
$\mathit{map}\,(\lambda x \to x + y)$	$\{\mathit{map}, y\}$
$(x + y, \lambda z \to z)$	$\{x, y\}$
$(x + y, \lambda x \to x)$	$\{x, y\}$

Le *renommage-*α n'affecte pas les variables libres

Portée dynamique

La *portée* est dite *dynamique* si elle est délimitée temporellement

La déclaration correspondant à un usage de variable est la plus *récente* déclaration du même identificateur encore active:

```
y = 1
f x = x + y
g n = (let y = f n in f y) + f (n + 1)
h m = (let y = g m in g y) + g (m + 1)
```

Un même usage de variable peut donc référer à différentes déclarations à différents moments

Propriétés de la portée dynamique

Accès direct aux variables “locales” de la fonction appelante!

Problème de *capture de nom*:

```
transX (x, vs)
  = map (\(a,b) -> (a + x, b)) vs
```

Le choix des identificateurs a de l’importance: pas de renommage-α!

En Emacs Lisp, on utilise une convention *{pkg}-{name}*

Pas de currying:

```
let f = \x -> \y -> x + y in f 1 2
```

Pourquoi portée dynamique

```
let dest = stdout          -- envoyer à stdout par défaut

    printfoo x = write dest (show x)

in ...

   let dest = open "foobar" in printfoo foo
```

Passage implicite d'arguments qui changent rarement

Implantation naïve dans un interpréteur

Pourquoi portée lexicale

Permet l'analyse statique (automatique ou humaine)

$\Rightarrow$ Mène à du code plus efficace

Liberté de choix des identificateurs

$$\texttt{let}\ x = 3\ \texttt{in}\ \mathit{foo}\ 0 + x$$

Est-ce correct de remplacer x par 3 ?

Est-ce correct ensuite d'éliminer la variable x ?

Fermetures, types, ordre d'évaluation

Fermetures

$\lambda x \to o\ x\ x$ a besoin d'un contexte qui défini o

Une fermeture associe une fonction à son environnement: $\lambda^E x \to e$

On distingue entre les expressions $\lambda x \to e$ et les valeurs $\lambda^E x \to e$

$$(E; \lambda x \to e) \quad \rightsquigarrow \quad (E; \lambda^E x \to e)$$
$$(E_2; (\lambda^{E_1} x \to e_1)\ e_2) \quad \rightsquigarrow \quad (E_1, x \mapsto e_2; e_1)$$

Seules les variables *libres* dans e sont nécessaires dans E

Fermetures comme objets

Les fermetures sont des *données*:

```
mkpair a b = \f -> if f then a else b
fst p = p true
snd p = p false
```

La fermeture “capture” les variables *libres* (ici, *a* et *b*)

Représentation en mémoire très raisonnable

Ordre d'évaluation

Choix principal: évaluer les arguments avant ou après l'appel

CBV = avant — *appel par valeur*

CBN = après — *appel par nom*

$$\texttt{let}\ x = f\ 0\ \texttt{in}\ g\ x$$

$$\texttt{where}\quad g\ x = x + x$$

$$\texttt{or}\quad g\ x = 1$$

Structures de données infinies

$$\mathit{zipWith} :: (\alpha \to \beta \to \gamma) \to [\alpha] \to [\beta] \to [\gamma]$$

$$\mathit{zipWith}\ f\ \mathit{xs}_1\ \mathit{xs}_2 = \mathit{map}\ (\mathit{uncurry}\ f)\ (\mathit{zip}\ \mathit{xs}_1\ \mathit{xs}_2)$$

$$\mathit{zipWith}\ f\ (x : xs)\ (y : ys) = f\ x\ y : \mathit{zipWith}\ f\ xs\ ys$$

$$\mathit{zipWith}\ f\ \_\ \_ = []$$

$$\mathit{ones} = 1 : \mathit{ones}$$

$$\mathit{numbers} = 0 : \mathit{zipWith}\ (+)\ \mathit{ones}\ \mathit{numbers}$$

$$\mathit{numbers} \equiv [0, 1, 2, 3, 4, \ldots]$$

Récursion cachée

Comment utiliser la récursion avec des fonctions anonymes?

```
fact' fact 0 = 1
fact' fact n = n * fact (n - 1)
fact = fact' fact
```

Il semble qu'on a seulement repoussé le problème. Sauf que *fact'* est n'a pas besoin de *fact* mais seulement de *fact'*:

```
fact' fact' 0 = 1
fact' fact' n = n * fact' fact' (n - 1)
fact = fact' fact'
```

Maintenant, on a un problème de *type*.

Concepts: Syntaxe

- Analyse lexicale, analyse syntaxique
- Backus-Naur Form
- Arbre de syntaxe abstraite
- Infixe/postfixe/préfixe
- Sucre syntaxique
- Notation sans sémantique

Concepts: Types

- Types primitifs
- Types produits
- Types somme
- Types récursifs
- Types paramétriques
- Interfaces, classes, signatures, contraintes
- Inférence de types
- Égalité

Concepts: Expressions

- Définitions locales
- Expressions conditionnelles
- Construction
- Filtrage
- Récursion

Concepts: Fonctions et variables

- Variables locales, portée
- Évaluation CBV, CBN, paresseuse
- Fonctions d'ordre supérieur
- Currying, fermetures

Concepts manquants

- Effets de bord, mutation
- Non-déterminisme
- Gestion mémoire
- Pointeurs, références
- Modularité, abstraction de donnée