Lambda-calcul et programmation

Jean-Jacques Lévy
INRIA

9-12-2009

CENTRE DE RECHERCHE COMMUN

INRIA
MICROSOFT RESEARCH

Plan

- Quelques langages de programmation:
  - Lambda calcul pur
  - PCF pur
  - PCF typé
  - Caml
  - Haskell
- Programmation fonctionnelle
- Impacts de la programmation fonctionnelle
- Futur

Lambda-calcul pur

CENTRE DE RECHERCHE COMMUN

INRIA MICROSOFT RESEARCH

Le langage

- Les lambda-expressions

M, N, P	$::=$	$x, y, z, \ldots$	(variables)
	$\mid$	$\lambda x.M$	(M fonction de x)
	$\mid$	$M\,(\,N\,)$	(M appliqué à N)

- Calculs "réductions"

$$(\lambda x.M)(N) \longrightarrow M\{x := N\}$$

Exemples de calcul

- Exemples

$$(\lambda x.x)N \longrightarrow N$$

$$(\lambda f.f\ N)(\lambda x.x) \longrightarrow (\lambda x.x)N \longrightarrow N$$

Exemples de calcul

- Exemples

$$(\lambda x.x)N \longrightarrow N$$

$$(\lambda f.f\ N)(\lambda x.x) \longrightarrow (\lambda x.x)N \longrightarrow N$$

$$(\lambda x.xx)(\lambda x.xN) \longrightarrow (\lambda x.xN)(\lambda x.xN) \longrightarrow (\lambda x.xN)N \longrightarrow NN$$

$$(\lambda x.xx)(\lambda x.xx) \longrightarrow (\lambda x.xx)(\lambda x.xx) \longrightarrow \cdots$$

Exemples de calcul

- Exemples

$$(\lambda x.x)N \longrightarrow N$$

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

$$(\lambda x.xx)(\lambda x.xN) \longrightarrow (\lambda x.xN)(\lambda x.xN) \longrightarrow (\lambda x.xN)N \longrightarrow NN$$

$$(\lambda x.xx)(\lambda x.xx) \longrightarrow (\lambda x.xx)(\lambda x.xx) \longrightarrow \cdots$$

$$Y_f = (\lambda x.f(xx))(\lambda x.f(xx)) \longrightarrow f((\lambda x.f(xx))(\lambda x.f(xx))) = f(Y_f)$$

$$f(Y_f) \longrightarrow f(f(Y_f)) \longrightarrow \cdots \longrightarrow f^n(Y_f) \longrightarrow \cdots$$

Calculer dans le lambda-calcul

- Booléens

$$\mathtt{Vrai} = \lambda x.\lambda y.x = K \qquad \mathtt{Vrai}\ M\ N \xrightarrow{\star} M$$
$$\mathtt{Faux} = \lambda x.\lambda y.y \qquad \mathtt{Faux}\ M\ N \xrightarrow{\star} N$$

- Paires et projections

$$\langle M, N \rangle = \lambda x.xMN \qquad \pi_1 \langle M, N \rangle \xrightarrow{\star} M$$
$$\pi_1 = \lambda x.x\,(\mathtt{Vrai}) \qquad \pi_2 \langle M, N \rangle \xrightarrow{\star} N$$
$$\pi_2 = \lambda x.x\,(\mathtt{Faux})$$

- Entiers ...

$$0 = \langle \mathtt{Vrai}, \mathtt{Vrai} \rangle \qquad \mathtt{Zero?}\ 0 \xrightarrow{\star} \mathtt{Vrai}$$
$$n + 1 = \langle \mathtt{Faux}, n \rangle \qquad \mathtt{Zero?}(n+1) \xrightarrow{\star} \mathtt{Faux}$$
$$\mathtt{Zero?} = \pi_1$$

Calculer dans le lambda-calcul

- ... Entiers

$$\texttt{Succ} = \lambda x. \langle \texttt{Faux}, x \rangle$$

$$\texttt{Pred} = \lambda x.\, \texttt{Zero?}\; x\; 0\; \pi_2$$

Calculer comme Church

- Lambda-*I* calcul

$\lambda x.M$ (*M* fonction non constante de *x*)

pas de $K = \lambda x.\lambda y.x$

- Entiers de Church

$n = \lambda f.\lambda x.f^n(x)$

$n \geq 1$

$n\, I \xrightarrow{*} I$

$I = \lambda x.x$

- Paires et projections

$\langle M, N\rangle = \lambda x.xMN$
$\pi_1 = \lambda p.p(\lambda x.\lambda y.\, y\, I\, x)$
$\pi_2 = \lambda p.p(\lambda x.\lambda y.\, x\, I\, y$

$\pi_1\langle m, n\rangle \xrightarrow{*} m$
$\pi_2\langle m, n\rangle \xrightarrow{*} n$

Calculer comme Church

- Lambda-*I* calcul

$\lambda x.M$ (*M* fonction non constante de *x*)

- Entiers de Church

$n = \lambda f.\lambda x.f^n(x)$

$n \geq 1$

$n\, I \xrightarrow{\star} I$

$I = \lambda x.x$

- Paires et projections

$\langle M, N \rangle = \lambda x.xMN$
$\pi_1 = \lambda p.p(\lambda x.\lambda y.\, y\, I\, x)$
$\pi_2 = \lambda p.p(\lambda x.\lambda y.\, x\, I\, y$

$\pi_1 \langle m, n \rangle \xrightarrow{\star} m$
$\pi_2 \langle m, n \rangle \xrightarrow{\star} n$

Calculer comme Church

- ... Entiers

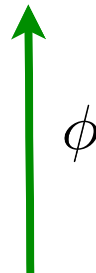
$$\texttt{Succ} = \lambda n.\lambda f.\lambda x.n\, f\,(f\, x)$$

$$\texttt{Pred} = \lambda n.\ \pi^3_3\ (n\,\phi\,\langle 1, 1, 1\rangle)$$

$$\phi = \lambda t.(\lambda x.\lambda y.\lambda z.\langle \texttt{Succ}\ x,\ x,\ y\rangle)(\pi^3_1\ t)(\pi^3_2\ t)(\pi^3_3\ t)$$

où $\pi^3_1,\ \pi^3_2,\ \pi^3_3$ sont les 3 projections sur les triplets

4	3	2
3	2	1
2	1	1
1	1	1



ϕ

registre à décalage! FIFO

Calculer comme Church

Alonzo Church

Stephen Kleene

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH

If L, M, N are formulas representing positive integers, then $2_1[M, N]$ conv M, $2_2[M, N]$ conv N, $3_1[L, M, N]$ conv L, $3_2[L, M, N]$ conv M, and $3_3[L, M, N]$ conv N.

Verification of this depends on the observation that, if M is a formula representing a positive integer, MI conv I (the mth power of the identity is the identity).

By the predecessor function of positive integers we mean the function whose value for the argument 1 is 1 and whose value for any other positive integer argument x is $x-1$. This function is λ-defined by

$$P \rightarrow \lambda a . 3_3(a(\lambda b[S(3_1 b), 3_1 b, 3_2 b])[1, 1, 1]).$$

For if K, L, M represent positive integers,

$$(\lambda b[S(3_1 b), 3_1 b, 3_2 b])[K, L, M] \text{ conv } [SK, K, L],$$

samedi 12 décembre 2009

mth power of the identity is the identity).

By the predecessor function of positive integers we mean the function whose value for the argument 1 is 1 and whose value for any other positive integer argument x is $x-1$. This function is λ-defined by

$$P \rightarrow \lambda a . \mathfrak{z}_3(a(\lambda b[S(\mathfrak{z}_1 b), \mathfrak{z}_1 b, \mathfrak{z}_2 b])[1, 1, 1]).$$

For if K, L, M represent positive integers,

$$(\lambda b[S(\mathfrak{z}_1 b), \mathfrak{z}_1 b, \mathfrak{z}_2 b])[K, L, M] \text{ conv } [SK, K, L],$$

and hence if A represents a positive integer,

$$A(\lambda b[S(\mathfrak{z}_1 b), \mathfrak{z}_1 b, \mathfrak{z}_2 b])[1, 1, 1] \text{ conv } [SA, A, B],$$

where B represents the predecessor of the positive integer represented by A. (The method of λ-definition of the predecessor function due to Kleene [35] is here modified by employment of a different formal representation of ordered triads.)

samedi 12 décembre 2009

2 + 5 = 7
6 − 3 =

PCF

Le langage

- Les termes

M, N, P	::=	$x, y, z, \ldots$	(variables)
	|	$\lambda x.M$	(M fonction de x)
	|	$M\,(\,N\,)$	(M appliqué à N)
	|	$\mathtt{n}$	(constante entière)
	|	$M \otimes N$	(opération arithmétique)
	|	$\mathtt{ifz}\ P\ \mathtt{then}\ M\ \mathtt{else}\ N$	(conditionnelle)

- Les calculs (“réductions”)

$$(\lambda x.M)(N) \longrightarrow M\{x := N\}$$

$$\underline{m} \otimes \underline{n} \longrightarrow \underline{m \otimes n}$$

$$\mathtt{ifz}\ \underline{0}\ \mathtt{then}\ M\ \mathtt{else}\ N \longrightarrow M$$

$$\mathtt{ifz}\ \underline{n+1}\ \mathtt{then}\ M\ \mathtt{else}\ N \longrightarrow N$$

Exemples de termes

$(\lambda x.\, x + 1)3 \longrightarrow 3 + 1 \longrightarrow 4$

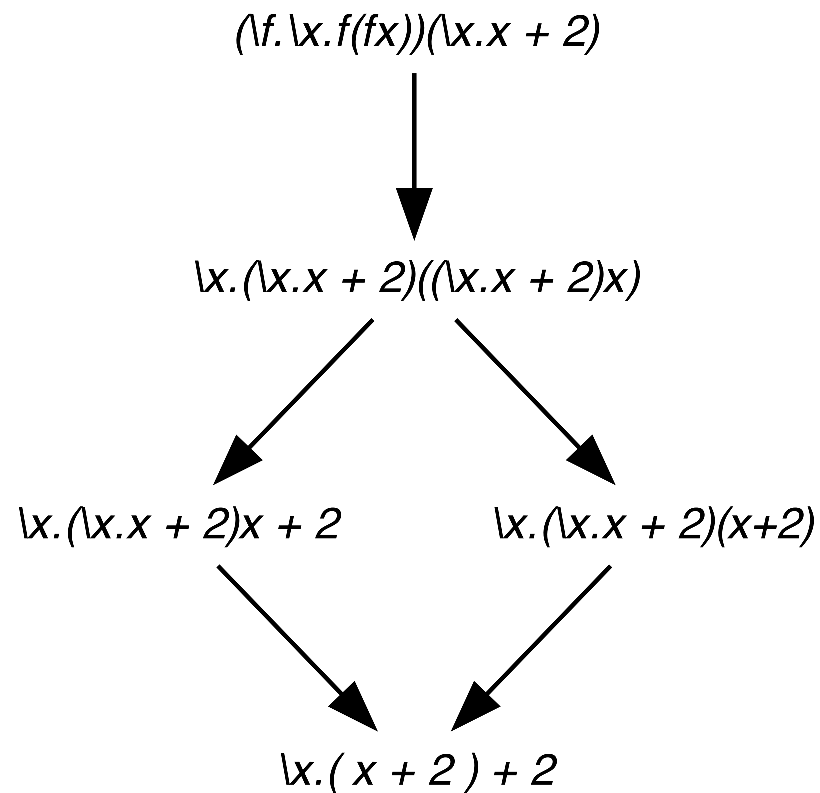
$(\lambda x.\, 2 * x + 2)4 \longrightarrow 2 * 4 + 2 \longrightarrow 8 + 2 \longrightarrow 10$

$(\lambda f.f3)(\lambda x.\, x + 2) \longrightarrow (\lambda x.\, x + 2)3 \longrightarrow 3 + 2 \longrightarrow 5$

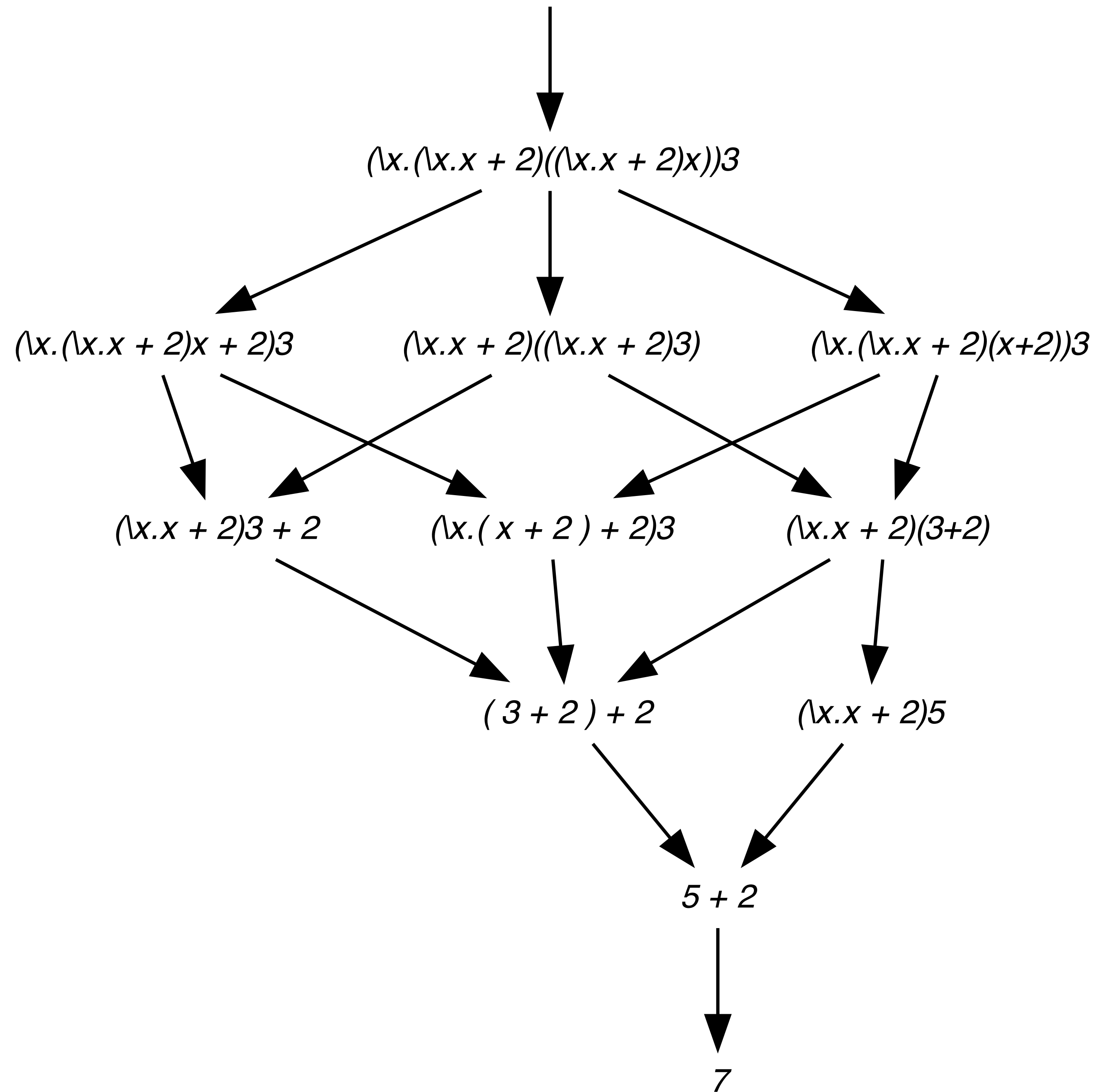
$(\lambda f.\lambda x.f(f\, x))(\lambda x.x + 2) \longrightarrow \ldots$

$$(\lambda f.\lambda x.f(f\,x))(\lambda x.\,x+2) \longrightarrow \ldots$$

$$(\lambda f.\lambda x.f(f\,x))(\lambda x.\,x+2) \longrightarrow \ldots$$



$(\lambda f.\lambda x.f(f\ x))(\lambda x.x+2)3 \longrightarrow \ldots$



Exemples de termes

$\texttt{Fact}(3)$

$\texttt{Fact} = Y(\lambda f.\lambda x.\ \texttt{ifz}\ x\ \texttt{then}\ 1\ \texttt{else}\ x \star f(x-1))$

$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

s'écrit

$(\lambda\,\texttt{Fact}\,.\,\texttt{Fact}(3))$

$(\ (\lambda Y.Y(\lambda f.\lambda x.\ \texttt{ifz}\ x\ \texttt{then}\ 1\ \texttt{else}\ x \star f(x-1)))$

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

(\Fact.Fact3)((\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf))

(\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf)3

(\f.Yf)(\f.\x.ifz x then 1 else x * f(x-1))3

(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))3

(\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))3

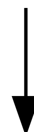
(\x.ifz x then 1 else x * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(x-1))3

ifz 3 then 1 else 3 * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(3-1)

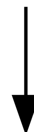
3 * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(3-1)

3 * (\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))(3-1)

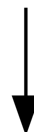
3 * (\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))(3-1)



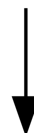
3 * (\x.ifz x then 1 else x * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(x-1))(3-1)



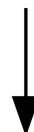
3 * (ifz 3 - 1 then 1 else (3-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))((3-1)-1))



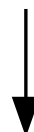
3 * (ifz 2 then 1 else (3-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))((3-1)-1))



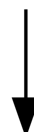
3 * ((3-1)*(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))((3-1)-1))



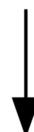
3 * (2*(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))((3-1)-1))



3 * (2*(\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))((3-1)-1))



3 * (2*(\x.ifz x then 1 else x * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(x-1))((3-1)-1))



3 * (2*(ifz (3-1) - 1 then 1 else ((3-1)-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(((3-1)-1)-1)))

3 * (2*(\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))((3-1)-1))

↓

3 * (2*(\x.ifz x then 1 else x * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(x-1))((3-1)-1))

↓

3 * (2*(ifz (3-1) - 1 then 1 else ((3-1)-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(((3-1)-1)-1)))

↓

3 * (2*(ifz 2 - 1 then 1 else ((3-1)-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(((3-1)-1)-1)))

↓

3 * (2*(ifz 1 then 1 else ((3-1)-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(((3-1)-1)-1)))

↓

3 * (2*(((3-1)-1)*(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(((3-1)-1)-1)))

↓

3 * (2*((2-1)*(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(((3-1)-1)-1)))

↓

3 * (2*(1*(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(((3-1)-1)-1)))

↓

3 * (2*(1*(\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))(((3-1)-1)-1)))

*3 * (2*(1*(\x.ifz x then 1 else x * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(x-1))(((3-1)-1)-1)))*

*3 * (2*(1*(ifz ((3-1)-1) - 1 then 1 else (((3-1)-1)-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))((((3-1)-1)-1)-1)*

*3 * (2*(1*(ifz (2-1) - 1 then 1 else (((3-1)-1)-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))((((3-1)-1)-1)-1))*

*3 * (2*(1*(ifz 1 - 1 then 1 else (((3-1)-1)-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))((((3-1)-1)-1)-1))))*

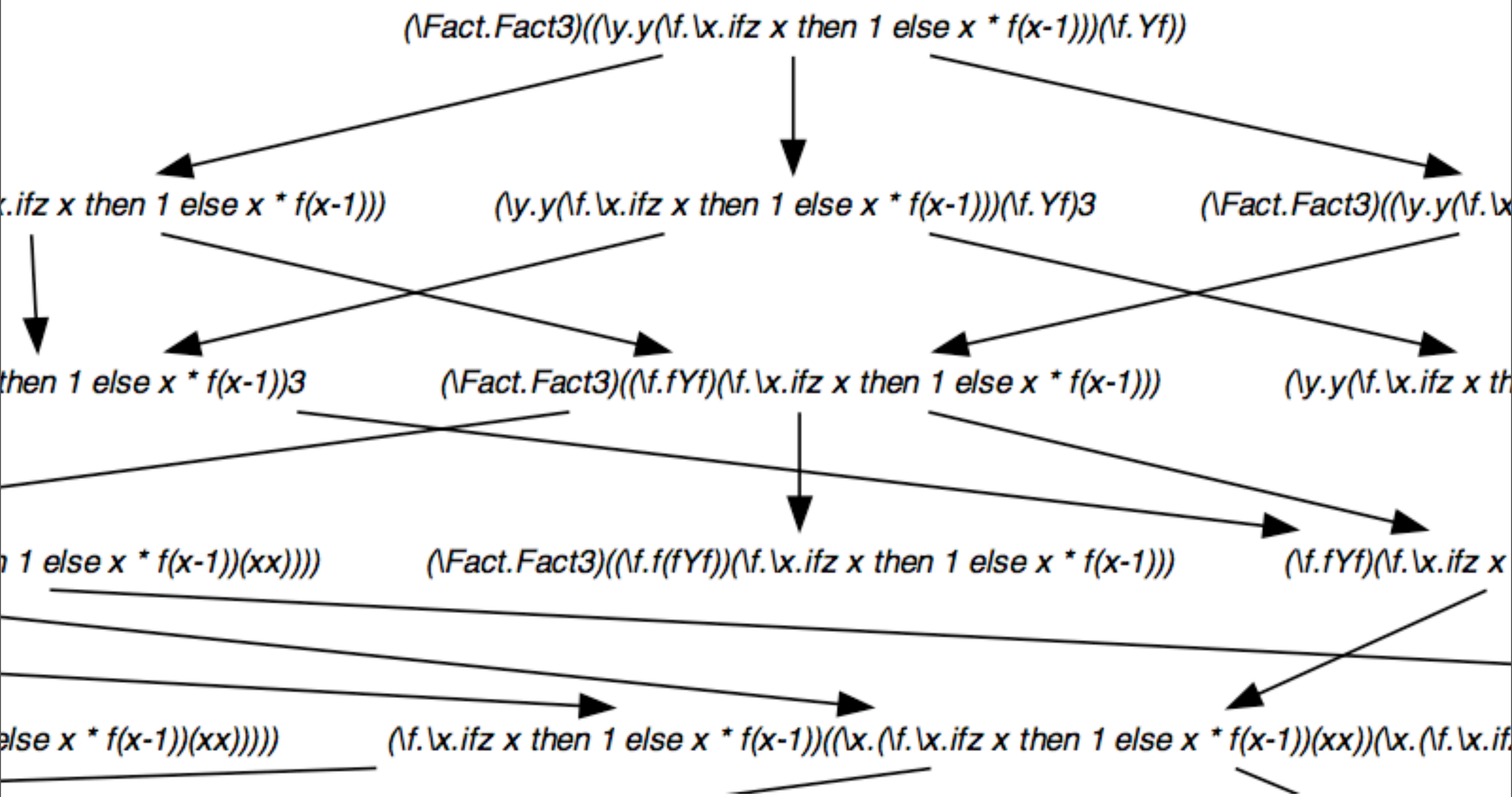
*3 * (2*(1*(ifz 0 then 1 else (((3-1)-1)-1) * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))((((3-1)-1)-1)-1))))*

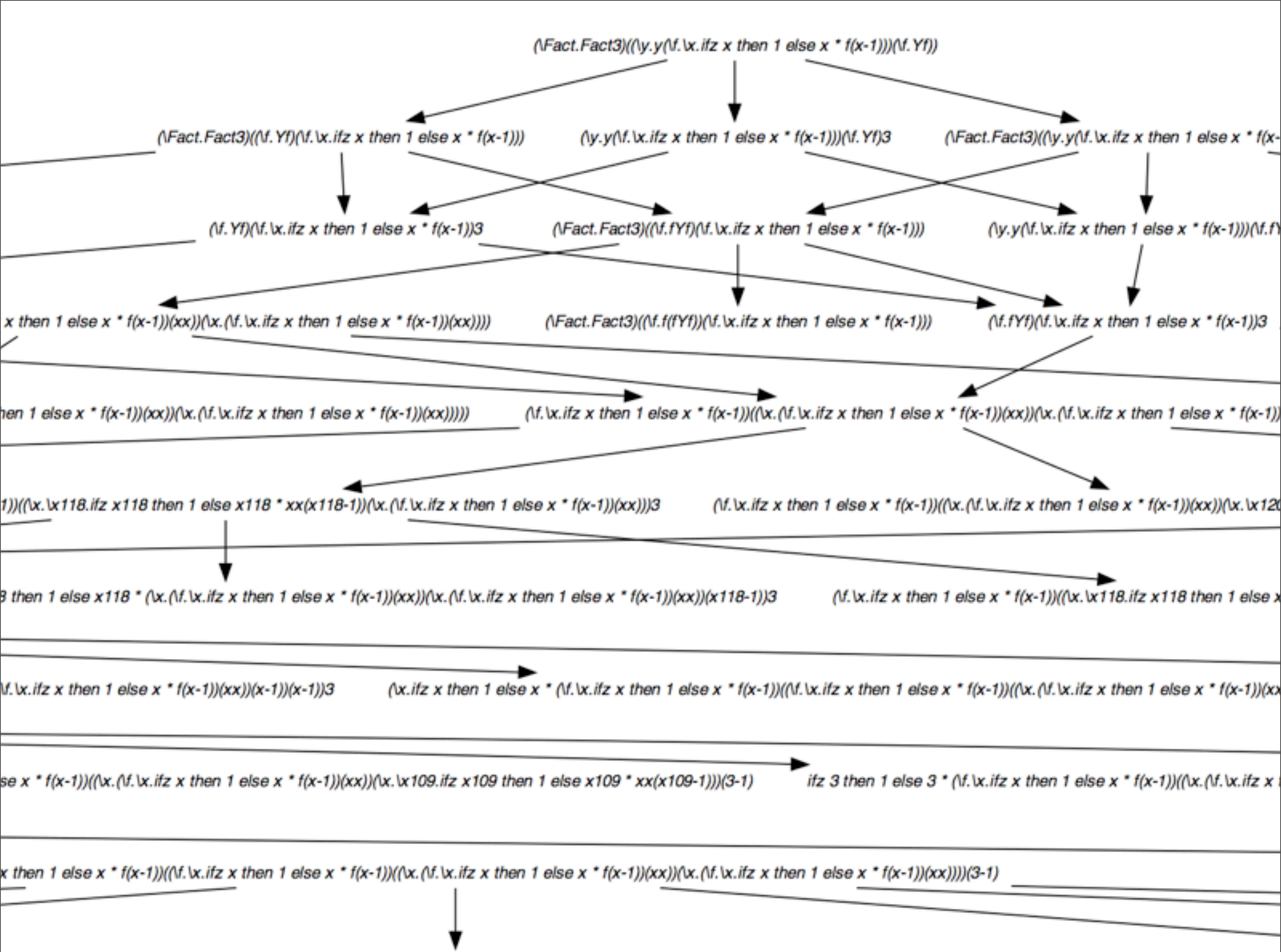
*3 * (2*(1*1))*

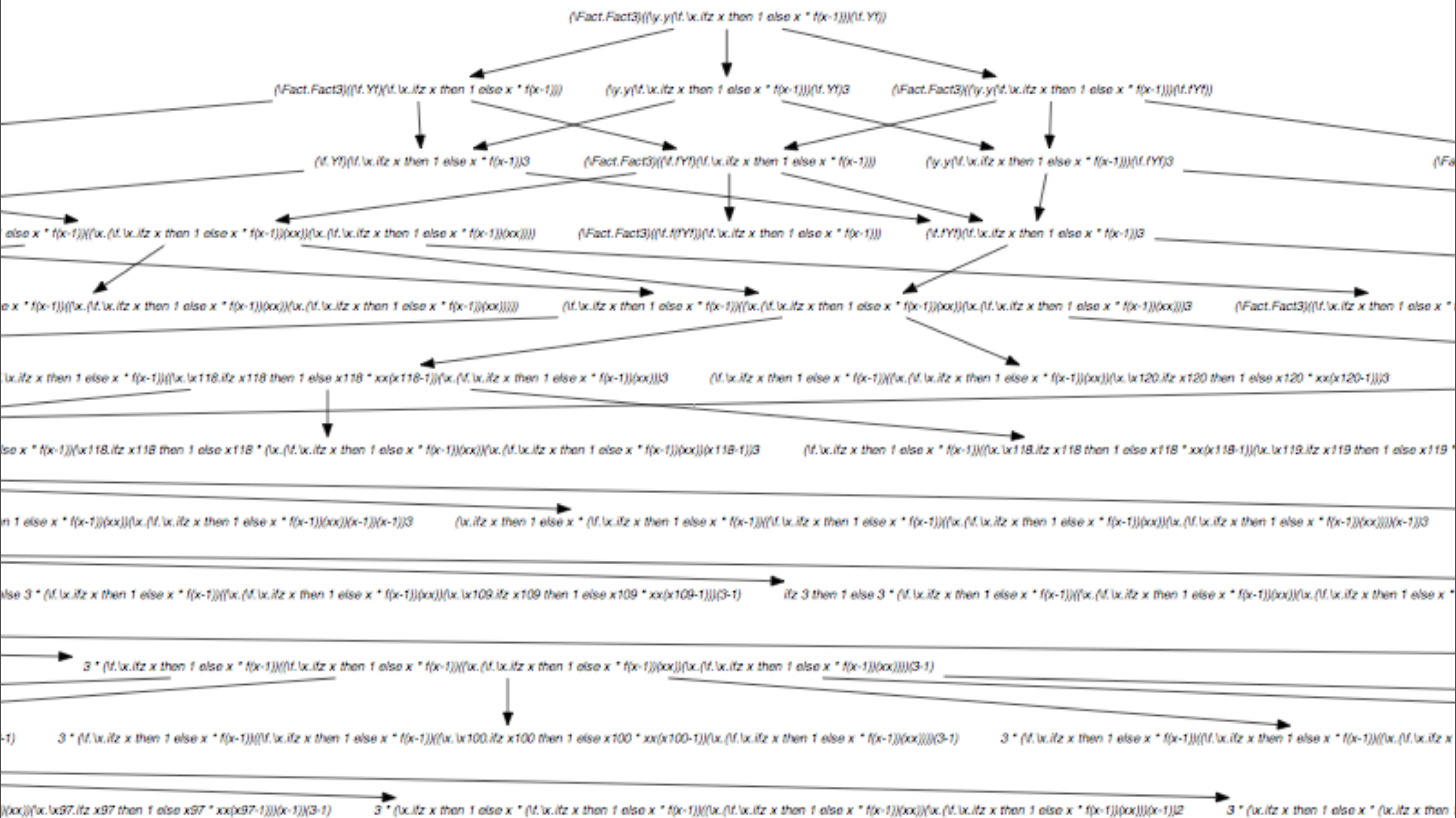
*3 * (2*1)*

*3 * 2*

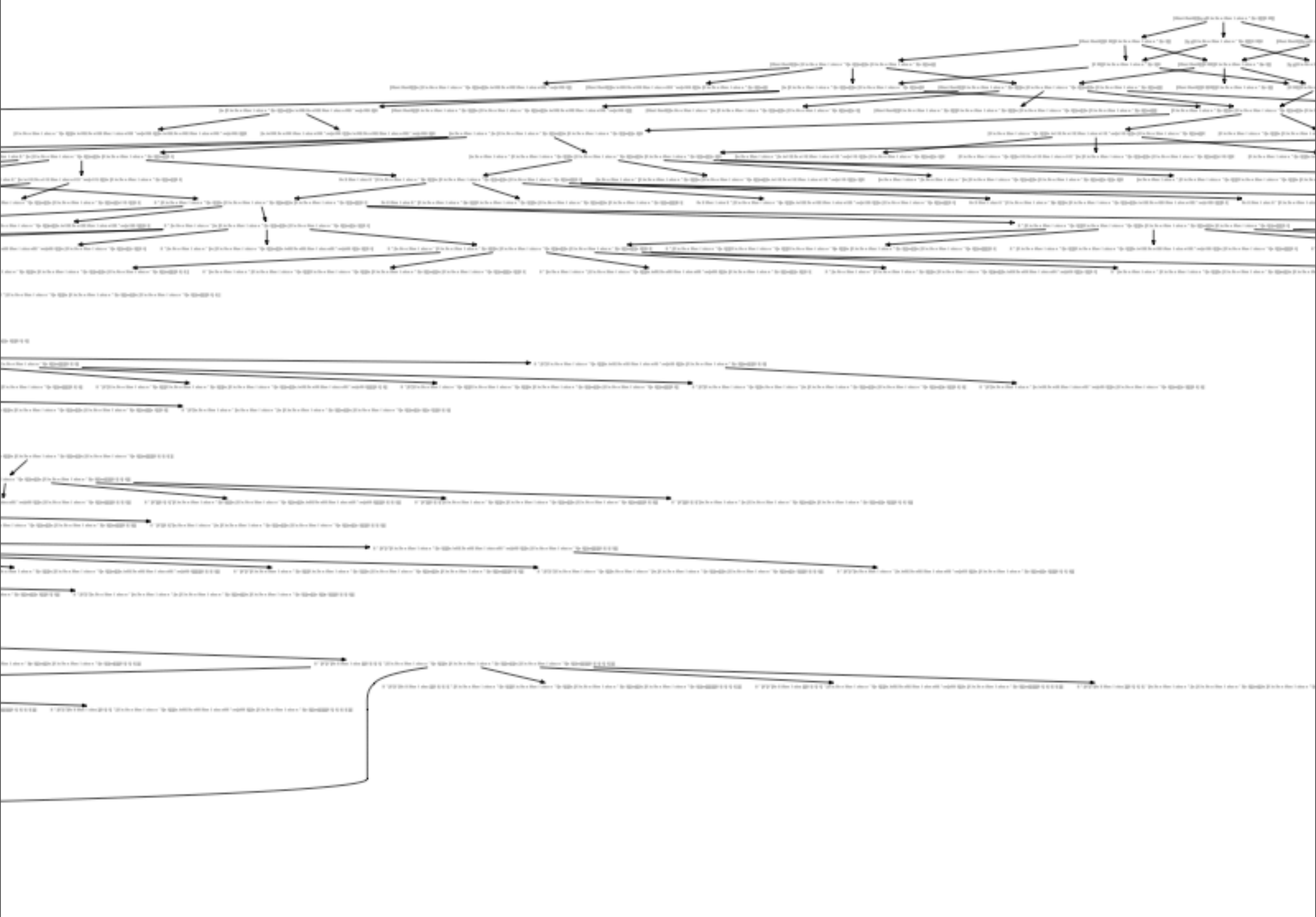
6

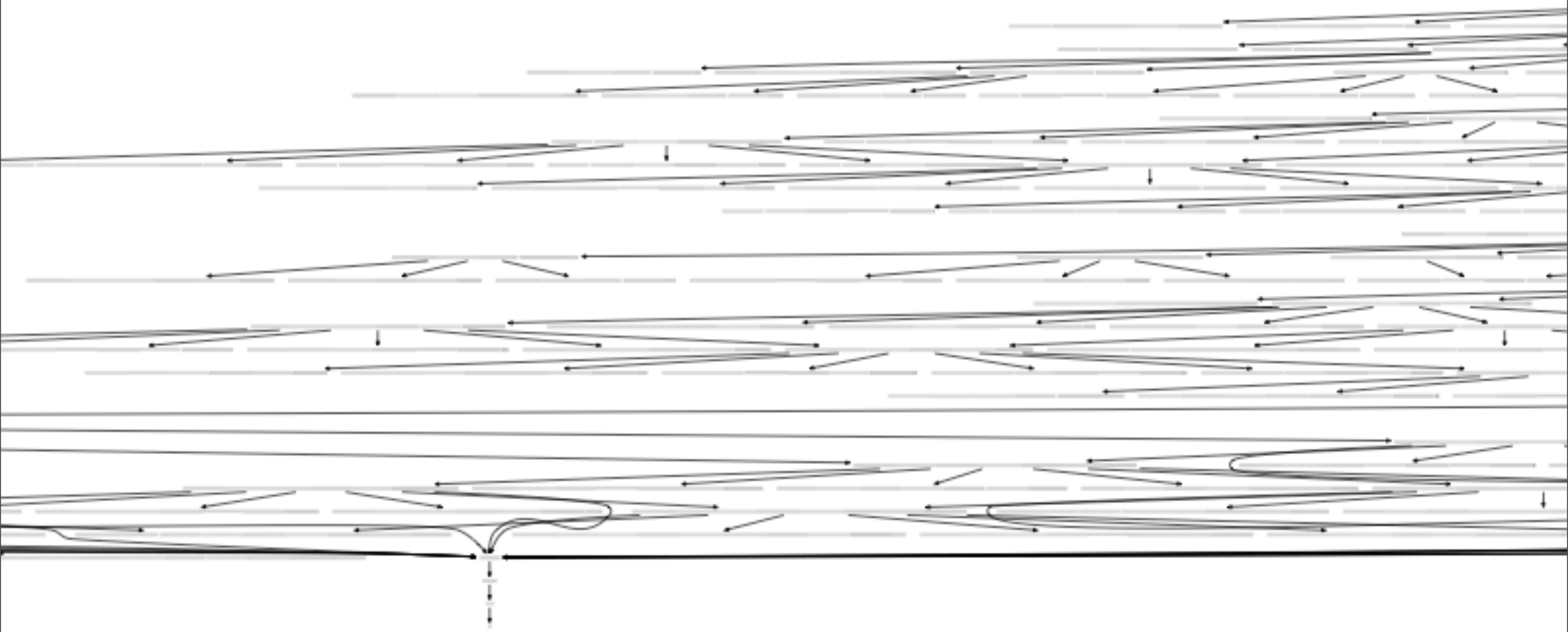

(\Fact.Fact3)((\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf))
.ifz x then 1 else x * f(x-1)))
(\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf)3
(\Fact.Fact3)((\y.y(\f.\x
then 1 else x * f(x-1))3
(\Fact.Fact3)((\f.fYf)(\f.\x.ifz x then 1 else x * f(x-1)))
(\y.y(\f.\x.ifz x th
1 else x * f(x-1))(xx))))
(\Fact.Fact3)((\f.f(fYf))(\f.\x.ifz x then 1 else x * f(x-1)))
(\f.fYf)(\f.\x.ifz x
else x * f(x-1))(xx)))))
(\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.if

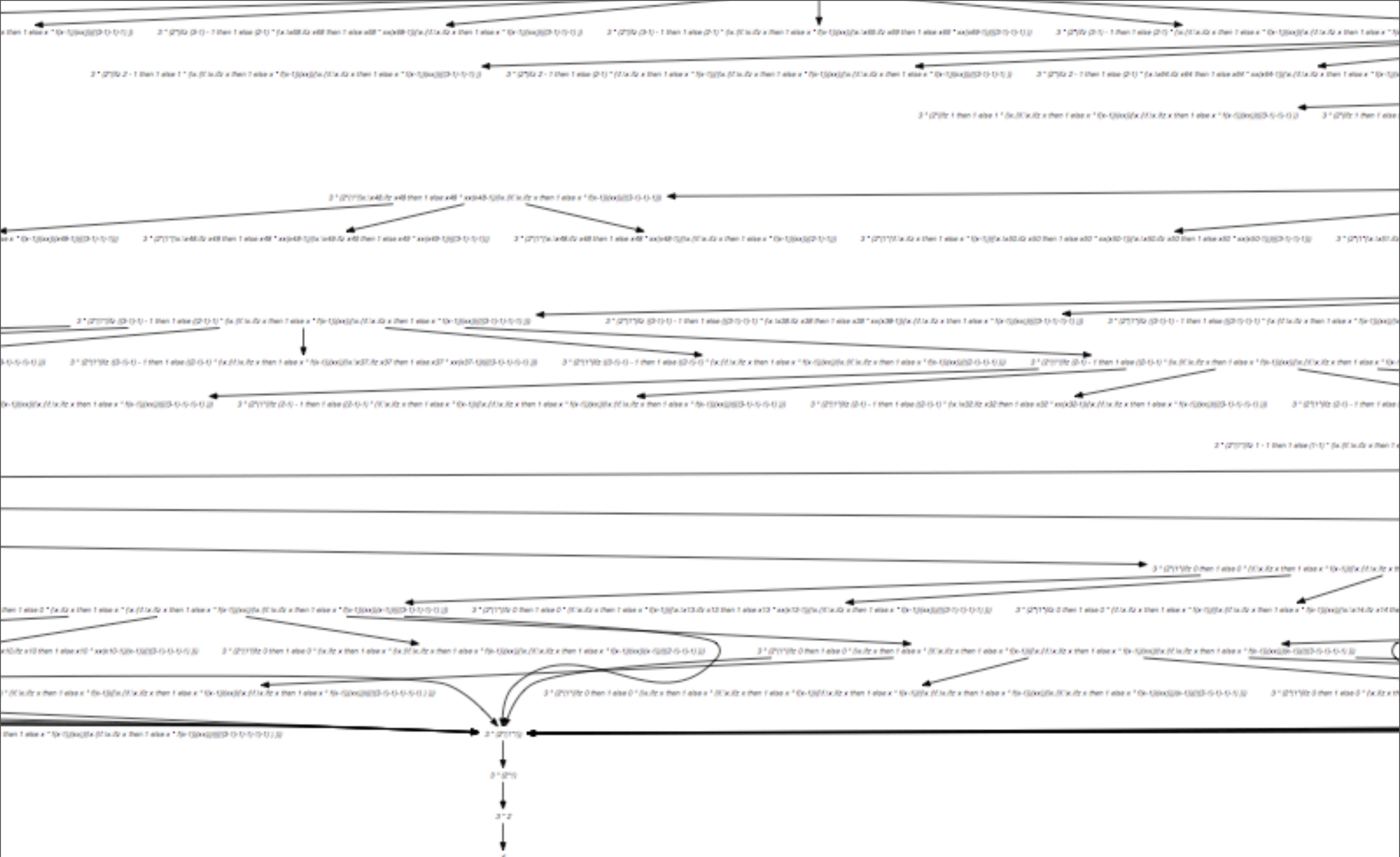

(\Fact.Fact3)((\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf))
(\Fact.Fact3)((\f.Yf)(\f.\x.ifz x then 1 else x * f(x-1)))
(\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf)3
(\f.Yf)(\f.\x.ifz x then 1 else x * f(x-1))3
(\Fact.Fact3)((\f.fYf)(\f.\x.ifz x then 1 else x * f(x-1)))
(\Fact.Fact3)((\f.f(fYf))(\f.\x.ifz x then 1 else x * f(x-1)))
(\f.fYf)(\f.\x.ifz x then 1 else x * f(x-1))3
x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))))
hen 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))))
1))((\x.\x118.ifz x118 then 1 else x118 * xx(x118-1))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))3
3 then 1 else x118 * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(x118-1))3
\f.\x.ifz x then 1 else x * f(x-1))(xx))(x-1))(x-1))3
se x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.\x109.ifz x109 then 1 else x109 * xx(x109-1)))(3-1)
ifz 3 then 1 else 3 * (\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x
x then 1 else x * f(x-1))((\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))))(3-1)



(\Fact.Fact3)((\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf))
(\Fact.Fact3)((\f.Yf)(\f.\x.ifz x then 1 else x * f(x-1)))
(\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.Yf)3
(\Fact.Fact3)((\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.fYf))
(\f.Yf)(\f.\x.ifz x then 1 else x * f(x-1))3
(\Fact.Fact3)((\f.fYf)(\f.\x.ifz x then 1 else x * f(x-1)))
(\y.y(\f.\x.ifz x then 1 else x * f(x-1)))(\f.fYf)3
(\Fact.Fact3)((\f.f(fYf))(\f.\x.ifz x then 1 else x * f(x-1)))
(\f.fYf)(\f.\x.ifz x then 1 else x * f(x-1))3
(\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))3
(\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.\x120.ifz x120 then 1 else x120 * xx(x120-1)))3
(\x.ifz x then 1 else x * (\f.\x.ifz x then 1 else x * f(x-1))((\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))))(x-1))3
3 * (\f.\x.ifz x then 1 else x * f(x-1))((\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))))(3-1)
3 * (\f.\x.ifz x then 1 else x * f(x-1))((\f.\x.ifz x then 1 else x * f(x-1))((\x.\x100.ifz x100 then 1 else x100 * xx(x100-1))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))))(3-1)
3 * (\x.ifz x then 1 else x * (\f.\x.ifz x then 1 else x * f(x-1))((\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx)))(x-1))2





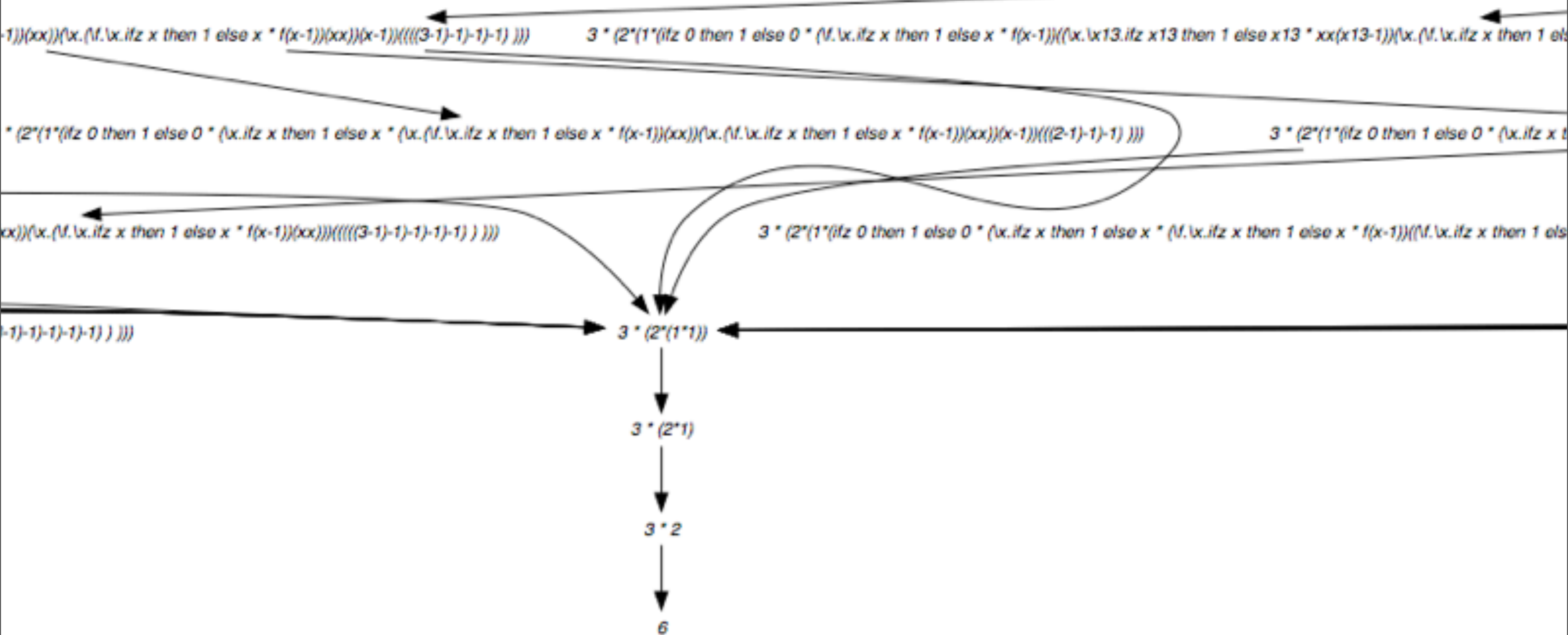

3 * (2*1)
3 * 2
6

3 * (2*(1*1))

3 * (2*1)

3 * 2

6


3 * (2*(1*1))
3 * (2*1)
3 * 2
6

\x.ifz x then 1 else x * f(x-1))(xx))(x-1))((((3-1)-1)-1)-1))))

3 * (2*(1*(ifz 0 then 1 else 0 * (\f.\x.ifz x then 1 else x * f(x-1))((\x.\x13.ifz x13 then

then 1 else 0 * (\x.ifz x then 1 else x * (\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(\x.(\f.\x.ifz x then 1 else x * f(x-1))(xx))(x-1))(((2-1)-1)-1))))

z x then 1 else x * f(x-1))(xx)))(((((3-1)-1)-1)-1)-1)))))

3 * (2*(1*(ifz 0 then 1 else 0 * (\x.ifz x then 1 else x * (\f.\x.i

))))

3 * (2*(1*1))

3 * (2*1)

3 * 2

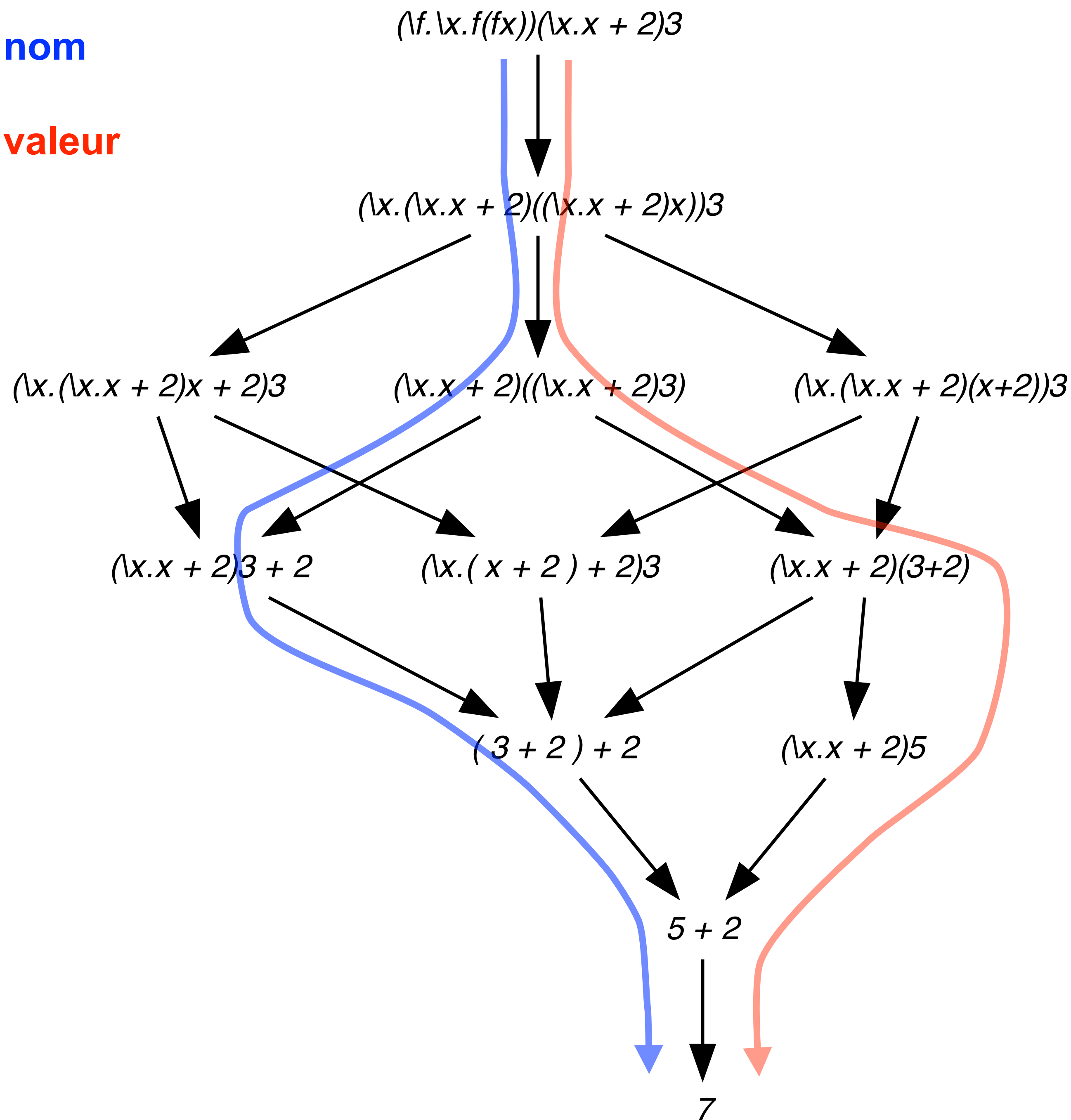
6

Propriétés

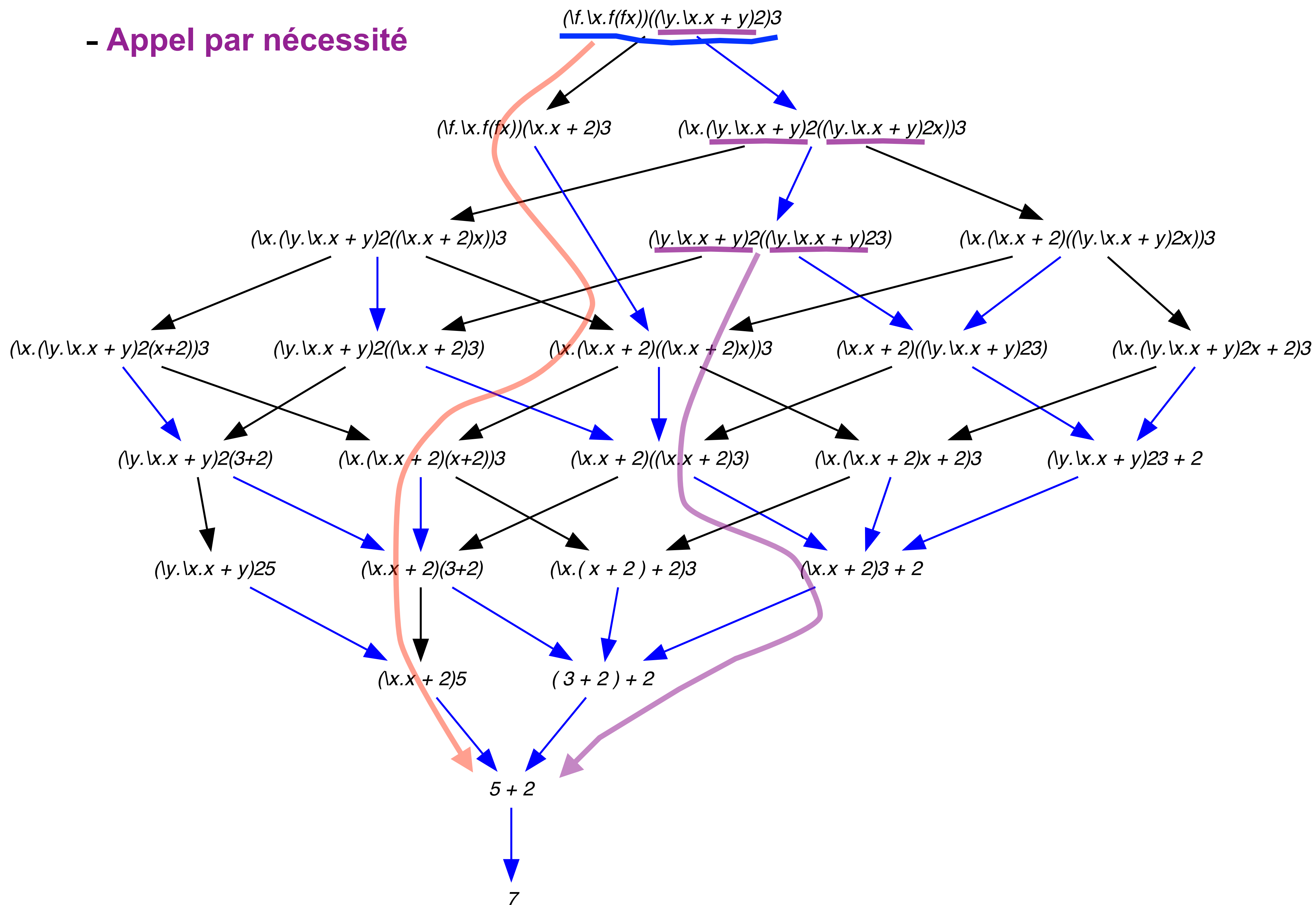
- Beaucoup de calculs différents
- Unicité du résultat (confluence -- Church Rosser)
- Plusieurs stratégies de réduction
  - **Appel par nom** (contracter le redex le + à gauche, le + externe)
  - **Appel par valeur** (on calcule d'abord la valeur des arguments)
  - **Appel par nécessité** (variante de l'appel par nom avec partage)

-Appel par nom

-Appel par valeur



$$(\lambda f.\lambda x.f(f\,x))((\lambda y.\lambda x.x + y)2)3 \rightarrow \ldots$$



Calcul par valeur

- Une valeur est tout terme **stable** par réduction et par substitution par des valeurs [i.e. un terme dont le résidu est toujours une valeur].
- Valeurs possibles:

$x, y, z, \ldots$	(variables)
$\lambda x.M$	(M fonction de x)
`n`	(constante entière)

Calcul par nécessité

- Le **partage** n’est pas simple dans le lambda-calcul, car il faut partager des fonctions !

- Le partage est facile à réaliser dans le lambda calcul dit faible où on ne réduit pas sous les lambda (ex: Haskell)

PCF typé

CENTRE DE RECHERCHE
COMMUN

INRIA
MICROSOFT RESEARCH

Le langage

- Les termes
- Les types

M, N, P	::=	$x, y, z, \ldots$	(variables)	$:\tau$
	|	$\lambda x.M$	(M fonction de x)	$:\sigma \to \tau$ si $x{:}\sigma$ $M{:}\tau$
	|	$M\,(\,N\,)$	(M appliqué à N)	$:\tau$ si $M{:}\sigma \to \tau$ $N{:}\sigma$
	|	$\mathtt{n}$	(constante entière)	$:\mathtt{int}$
	|	$M \otimes N$	(opération arithmétique)	$:\mathtt{int}$ si $M, N{:}\ \mathtt{int}$
	|	$\mathtt{ifz}\ P\ \mathtt{then}\ M\ \mathtt{else}\ N$	(conditionnelle)	$:\tau$ si $P{:}\ \mathtt{int}$ $M, N{:}\tau$
	|	Y	(récursion)	$:(\tau \to \tau) \to \tau$

- Les calculs ("réductions")

$$(\lambda x.M)(N) \longrightarrow M\{x := N\}$$

$$\underline{m} \otimes \underline{n} \longrightarrow \underline{m \otimes n}$$

$$\mathtt{ifz}\ \underline{0}\ \mathtt{then}\ M\ \mathtt{else}\ N \longrightarrow M$$

$$\mathtt{ifz}\ \underline{n+1}\ \mathtt{then}\ M\ \mathtt{else}\ N \longrightarrow N$$

$$YM \longrightarrow M(YM)$$

Intérêt du typage

- Jamais

+ = ???

- Jamais

$3 + \lambda x.x$ $\quad 4(5)$ $\quad 20(\lambda x.x)$ $\quad \texttt{ifz}\ \lambda x.x\ \texttt{then}\ 1\ \texttt{else}\ 3$

$\lambda x.xx$ $\quad \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Intérêt du typage

- Jamais

$3 + \lambda x.x$ $\quad 4(5)$ $\quad 20(\lambda x.x)$ $\quad \texttt{ifz}\ \lambda x.x\ \texttt{then}\ 1\ \texttt{else}\ 3$

$\lambda x.xx$ $\quad \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Intérêt du typage

- Jamais

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

Intérêt du typage

Intérêt du typage

- Jamais de bogues de typage à l’exécution
- termes **plus naturels**
- PCF typé non récursif (sans *Y*) est fortement normalisable [tous ses **calculs terminent**]
- il faut un opérateur de récursion Y pour exprimer autant que PCF pur

Normalisation forte

- pourquoi PCF typé non récursif termine ?

$$(\lambda x.\cdots xN\cdots)(\lambda y.M) \longrightarrow \cdots(\lambda y.M)N'\cdots$$

$\sigma \to \tau$; σ ; σ

crée

$$(\lambda x.\lambda y.M)NP \longrightarrow (\lambda y.M')P$$

τ ; $\sigma \to \tau$; τ

crée

- + théorème des développements finis

ML

CENTRE DE RECHERCHE COMMUN

INRIA MICROSOFT RESEARCH

Le langage

- initialement le méta-langage de LCF (*logic for computable functions*) `[robin milner, 1978]`
- PCF typé avec des types polymorphes
- types de données récursifs (listes, arbres, etc)
- filtrage (*pattern matching*)
- inférence de type
- quelques données sont modifiables : références, tableaux, champs d'enregistrements, ...

Le langage

- Les termes
- Les types

M, N, P	::=	$x, y, z, \ldots$	(variables)	$:\sigma\{\vec{\alpha} := \vec{\tau}\}$ si $x:\forall\vec{\alpha}\,\sigma$
	|	$\lambda x.M$	(M fonction de x)	$:\sigma \to \tau$ si $x:\sigma$ $M:\tau$
	|	$M\,(\,N\,)$	(M appliqué à N)	$:\tau$ si $M:\sigma \to \tau$ $N:\sigma$
	|	$\mathtt{n}$	(constante entière)	$:\mathtt{int}$
	|	$M \otimes N$	(opération arithmétique)	$:\mathtt{int}$ si $M, N: \mathtt{int}$
	|	$\mathtt{ifz}\ P\ \mathtt{then}\ M\ \mathtt{else}\ N$	(conditionnelle)	$:\tau$ si $P: \mathtt{int}$ $M, N:\tau$
	|	Y	(récursion)	$:(\tau \to \tau) \to \tau$
	|	$\mathtt{let}\ x = M\ \mathtt{in}\ N$	(définition polymorphe)	$:\tau$ si $x:\forall\vec{\alpha}\,\sigma$ $M:\sigma$

- Les calculs (“réductions”)

mêmes règles que pour PCF typé + règle suivante

$$\mathtt{let}\ x = M\ \mathtt{in}\ N \longrightarrow N\{x := M\}$$

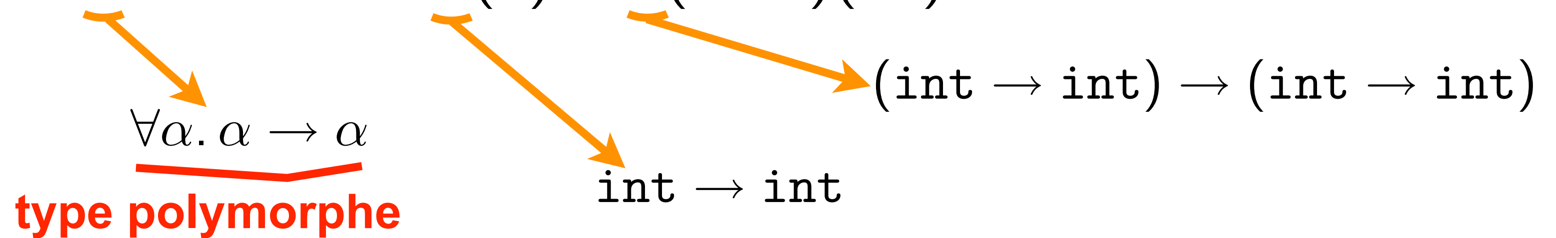
Le langage

- L'identité a le type suivant

$$\lambda x.x : \alpha \to \alpha$$

- Donc, l'expression suivante est typée

$$\texttt{let}\ f = \lambda x.x\ \texttt{in}\ f(3) + f(succ)(10)$$



où

$$succ = \lambda x.x + 1 : \texttt{int} \to \texttt{int}$$

Le langage

- Ne pas confondre $\texttt{let}\, x = M \texttt{ in } N$ et $(\lambda x.N)M$:

$$\texttt{let}\, f = \lambda x.x \texttt{ in } f(3) + f(succ)(10)$$

est typé,

$$(\lambda f.\, f(3) + f(succ)(10))(\lambda x.x)$$

n'est pas typé.

- Dans une fonction, toutes les occurences de l'argument ont un même type.

Le langage

- Tout terme de ML a un type principal (le + général)
- L'algorithme Damas-Milner, fondé sur **l'unification** de termes de 1er ordre, calcule ce type principal.
- La complexité de cet algorithme est super exponentielle, mais quasi linéaire en pratique.
- En ML, il y a donc inférence de type. Pas la peine d'entrer les types des termes manuellement.

Le langage

- Listes, tableaux, enregistrements polymorphes prédéfinis.
- On définit aussi des types récursifs polymorphes à partir de **constructeurs** et autres types par **sommation** ou **produit**.
- Le filtrage permet d'accéder facilement aux arguments des constructeurs. Modèles du filtrage == Destructeurs

→ Ocaml

Ocaml

CENTRE DE RECHERCHE
COMMUN

INRIA
MICROSOFT RESEARCH

Le langage

- Un **vrai** langage de programmation (pas de lettres grecques...)
- version INRIA de ML `[xavier leroy, et al]`
- a aussi des données modifiables (références, tableaux, chaînes de caractères, enregistrements)
- stratégie d'évaluation en **appel par valeur**
- **modules pratiques** et compréhensibles
- a une partie orientée-objets
- très bon environnement de programmation
- simple et très efficace
- plutôt sophistiqué comme premier langage

Le langage

- L'identité a le type suivant

```
# function x -> x ;;
- : 'a -> 'a = <fun>
```

$\longleftrightarrow$ $\lambda x.x$

$\longleftrightarrow$ $\alpha \rightarrow \alpha$

- Donc, l'expression suivante est typée

```
# let f = function x -> x in
  f(3) + f(succ)(10) ;;
  - : int = 14
```

$\forall \alpha. \alpha \rightarrow \alpha$
type polymorphe

$\texttt{int} \rightarrow \texttt{int}$

$(\texttt{int} \rightarrow \texttt{int}) \rightarrow (\texttt{int} \rightarrow \texttt{int})$

où

```
# let succ = function x -> x + 1 ;;
val succ : int -> int = <fun>
```

Le langage

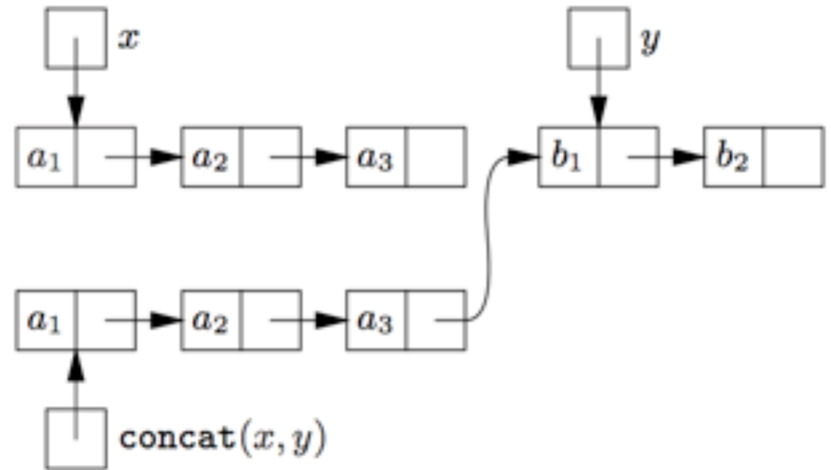
- Ne pas confondre `let` $x = M$ `in` N et $(\lambda x.N)M$:

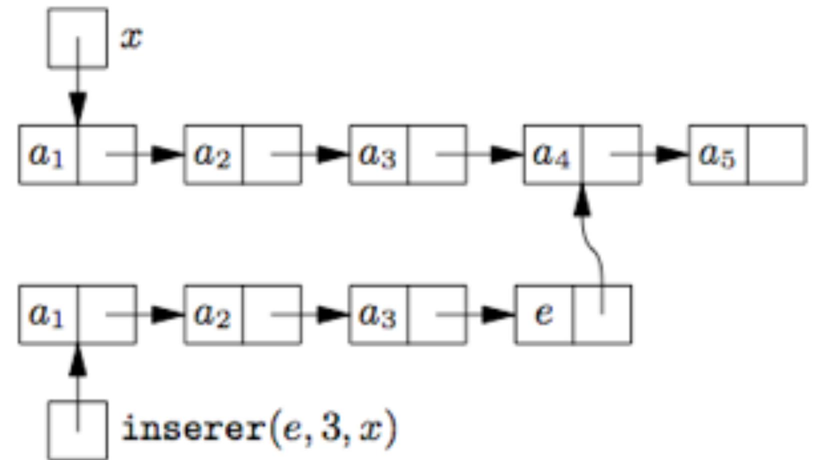
```
# (function f -> f(3) + f(succ)(10)) (function x -> x) ;;
Characters 22-23:
  (function f -> f(3) + f(succ)(10)) (function x -> x) ;;
                        ^
Error: This function is applied to too many arguments;
maybe you forgot a `;'
```

Filtrage

- Filtrage sur les **listes**

```
# let rec length x = match x with
  | [ ] -> 0
  | a :: x' -> 1 + length x' ;;
    val length : 'a list -> int = <fun>
# let rec concat x y = match x with
  | [ ] -> y
  | a :: x' -> a :: concat x' y ;;
    val concat : 'a list -> 'a list -> 'a list = <fun>
# let rec reverse x = match x with
  | [ ] -> [ ]
  | a :: x' -> concat (reverse x') [a] ;;
    val reverse : 'a list -> 'a list = <fun>
# let rec insert a n x =
    if n = 0 then a :: x else match x with
    | [ ] -> [ ]
    | b :: x' -> b :: insert a (n-1) x' ;;
      val insert : 'a -> int -> 'a list -> 'a list
```





- opérations avec types **polymorphes**
- fonctions **non destructives** : arguments non modifiés.

Filtrage

- **récurrence structurelle** == récursion + filtrage

```
type term = Var of string | Const of int | Plus of term * term | Minus of term * term
         | Mult of term * term | Ifz of term * term * term;;

type envt = (string * term) list;;

let rec eval t e = match t with
  | Var(x) -> List.assoc x e
  | Const(n) -> n
  | Plus(t1, t2) -> (eval t1 e) + (eval t2 e)
  | Minus(t1, t2) -> (eval t1 e) - (eval t2 e)
  | Mult(t1, t2) -> (eval t1 e) * (eval t2 e)
  | Ifz(p, t1, t2) -> if (eval p e = 0) then eval t1 e else eval t2 e ;;

let t = Minus(Plus(Mult( Const(3), Var ("x")), Var("y")),
              Mult(Const(2), Var("z")));;

let e = [("x", 45); ("y", 22); ("z", 17)] ;;
```

Filtrage

- les **fonctions** sont des valeurs comme les autres

```
# let a = Array.init 10 (function i -> i * i) ;;

# let b = Array.init 10 (function i -> Random.int 40) ;;

# let min = function x -> function y -> if x < y then x else y ;;

# let valeurMinDe a = Array.fold_left min max_int a ;;

# valeurMinDe b ;;
```

Filtrage

- les fonctions sont des valeurs comme les autres

```
# let a = Array.init 10 (function i -> i * i) ;;
val a : int array = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
# let b = Array.init 10 (function i -> Random.int 40) ;;
val b : int array = [|34; 22; 4; 18; 36; 2; 20; 10; 24; 1|]
# let min = function x -> function y -> if x < y then x else y ;;
val min : 'a -> 'a -> 'a = <fun>
# let valeurMinDe a = Array.fold_left min max_int a ;;
val valeurMinDe : int array -> int = <fun>
# valeurMinDe b ;;
```

Modules

- Types abstraits

```
module Fifo : sig
    type 'a t
    exception Empty_Fifo
    val create : unit -> 'a t
    val add : 'a t -> 'a -> unit
    val take : 'a t -> 'a
    val iter : ('a -> unit) -> 'a t -> unit
end = struct
  type 'a t = {mutable hd : 'a list; mutable tl: 'a list }
  (* hd points to fst of queue; tl points to last of queue *)
  exception Empty_Fifo
  let create () = {hd = [ ]; tl = [ ] }
  let add  f x = match f.hd, f.tl with
    | [ ], [ ] -> f.tl <- [x]; f.hd <- f.tl
    | _ , _ -> f.tl <- [x]; f.hd <- f.hd @ f.tl
  let take f = match f.hd with
    | [ ] -> raise Empty_Fifo
    | [ x ] -> f.hd <- [ ]; f.tl <- [ ]; x
    | x :: f' -> f.hd <- f'; x
  let iter f fifo = List.iter f fifo.hd
end ;;
```

signature

implémentation

- **encapsulation** des implémentations
- compositionalité

Haskell

CENTRE DE RECHERCHE
COMMUN

INRIA
MICROSOFT RESEARCH

Le langage

- Un autre **vrai** langage de programmation
- version GHC `[simon peyton-jones, et al]`
- sépare les données modifiables (références, tableaux, chaînes de caractères, enregistrements) dans des **monades**
- les monades peuvent utiliser tous les termes, mais seuls les monades peuvent appeler des monades
- stratégie d'évaluation en **appel par nom** (langage paresseux)
- permet la **surcharge** avec des *type-classes*
- très bon environnement de programmation
- un peu complexe

Programmation fonctionnelle

CENTRE DE RECHERCHE COMMUN

INRIA MICROSOFT RESEARCH

Langages fonctionnels

- Principalement des expressions:
  - PCF, booléens, n-uplets, listes, arbres, tableaux, records, chaînes
- Aussi des instructions qui modifient la mémoire:
  - affectations, données modifiables (tableaux, records, chaînes), entrées/sorties
- En Haskell, on sépare expressions et instructions avec les ``monades''.
- Principe des langages fonctionnels:

Réduire le nombre d'effets de bord dans les programmes

- avec un minimum d'états mémoire et de données modifiables
- Les types servent à identifier les données modifiables

Langages fonctionnels

- Un langage fonctionnel par nom (Haskell) vérifie:

$$(\lambda x.M)N \;\equiv\; M\{x := N\}$$

- Un langage fonctionnel par valeur (Caml) vérifie:

$$(\lambda x.M)V \;\equiv\; M\{x := V\}$$

- Des optimisations comme le partage des sous-expressions communes ou l'évaluation parallèle sont valides
- Le raisonnement équationnel est aussi valide
- La programmation devient plus sûre

Impact de la programmation fonctionelle

CENTRE DE RECHERCHE
COMMUN

INRIA
MICROSOFT RESEARCH

Quelques succès

- F# dans Visual Studio
- generics de **.NET**, programmation fonctionnelle populaire chez les programmeurs .NET
- generics, queries, lambdas, type inference, expression-tree meta-programming dans C#
- FP for "data rich" problems (e.g. statistical machine learning or symbolic programming) or "control rich" problems (e.g. parallel and asynchronous programming)
- Ocaml: Jane Street Capital [société d'arbitrage à New-York et Londres]
- F# a gagné le concours *adPredict*
- ``There is essentially near-universal agreement in industry and academia that elements of functional programming (**immutability**, queries, task **parallelism**, controlling **side-effects**, **data parallelism**, functional reactive programming) are essential in applied parallel and concurrent programming both in the short, medium and long terms. This plays out in different ways in different languages and tool chains '' [don syme - MSRC]
- **Erlang**: Ericsson, programmes dans les **téléphones**
- Ensemble, Cornell: bibliothèques d'algorithmes distribués programmée en Ocaml
- MIT: Leiserson's **FFTW** (la transformée de Fourier la plus rapide de l'Ouest) project using ML code to generate extremely optimized c code for dft's tailored to the machine architecture

Quelques succès

- coq, twelf, hol, isabelle, hol-light, bonne partie de nuprl sont écrits en ML. (**outils** puissants **de vérification** avec déjà de belles applications)
- une bonne partie des **compilateurs** modernes
- sql a un noyau fonctionnel
- tous les applications de Galois Connections sont écrites en Haskell (analyse de prog.)
- Bluespec (Arvind) est un langage fonctionnel pour **décrire le hardware**.
- **unison** (le meilleur synchroniseur de fichiers) est écrit en Ocaml
- beaucoup des problèmes de la **programmation objet** ont des solutions proches de celles pour les langages fonctionnels
- Frama-C (CEA, flottant), **Astrée**, l'analyseur statique pour C (airbus A380)
- IABC, le vieil analyseur d'**alias** d'**Alain Deutsch** (Ariane 502)
- et ... tous les autres de profundis par ma faute

Conclusion

CENTRE DE RECHERCHE
COMMUN

INRIA
MICROSOFT RESEARCH

Conclusion et Futur

- La longue route du lambda-calcul aux langages modernes !!
  a démontré l'intérêt du **typage fort** et **expressif**

- en route vers le **logiciel certifié**
  [beaucoup plus dur que le hardware certifié]
- des progrès en cours sur les types [système F, ou plus haut dans la hiérarchie des types]
- refinement types avec **formules logiques** (et puissants SMTs)
- langages pour programmer les *Data Centers*
- etc...

Slogans de fin

Le lambda-calcul est **propre** et **expressif**

Les langages de programmation issus du lambda-calcul sont **propres** et **expressifs**

Références

Alonzo Church, The calculi of Lambda-Conversion, AMS studies 6, Princeton University Press, 1941.

Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157-166, March 1966.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348-375, August 1978.

Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223-255, 1977.

Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125-159, 1975.

CENTRE DE RECHERCHE
COMMUN

INRIA
MICROSOFT RESEARCH