Quick Calculation

$A \text{ supports } (\lambda x.x \notin A)$

Quick Calculation

$$
\begin{aligned}
& A \; supports \; (\lambda x. x \notin A) \\
\stackrel{\text{def}}{=} \; & \forall a\, b.\; a, b \notin A \Rightarrow \\
& \qquad (a\, b) \bullet (\lambda x. x \notin A) = (\lambda x. x \notin A)
\end{aligned}
$$

Quick Calculation

$$A \ supports\ (\lambda x. x \notin A)$$

$$\overset{\text{def}}{=} \forall a\ b.\ a, b \notin A \Rightarrow (a\,b) \bullet (\lambda x. x \notin A) = (\lambda x. x \notin A)$$

$$= \forall a\ b.\ a, b \notin A \Rightarrow (\lambda x. (a\,b) \bullet ((a\,b) \bullet x \notin A)) = (\lambda x. x \notin A)$$

Quick Calculation

$$
\begin{aligned}
& A \ supports \ (\lambda x.x \notin A) \\
\overset{\text{def}}{=}\ & \forall a\ b.\ a, b \notin A \Rightarrow \\
& \qquad (a\,b)\bullet(\lambda x.x \notin A) = (\lambda x.x \notin A) \\
=\ & \forall a\ b.\ a, b \notin A \Rightarrow \\
& \qquad (\lambda x.(a\,b)\bullet((a\,b)\bullet x \notin A)) = (\lambda x.x \notin A) \\
=\ & \forall a\ b.\ a, b \notin A \Rightarrow \\
& \qquad (\lambda x.((a\,b)\bullet(a\,b)\bullet x) \notin (a\,b)\bullet A) = (\lambda x.x \notin A)
\end{aligned}
$$

Quick Calculation

$$
\begin{aligned}
& A \; supports \; (\lambda x.x \notin A) \\
\overset{\text{def}}{=} \; & \forall a \; b. \; a, b \notin A \Rightarrow \\
& \qquad (a\,b)\bullet(\lambda x.x \notin A) = (\lambda x.x \notin A) \\
= \; & \forall a \; b. \; a, b \notin A \Rightarrow \\
& \qquad (\lambda x.(a\,b)\bullet((a\,b)\bullet x \notin A)) = (\lambda x.x \notin A) \\
= \; & \forall a \; b. \; a, b \notin A \Rightarrow \\
& \qquad (\lambda x.((a\,b)\bullet(a\,b)\bullet x) \notin (a\,b)\bullet A) = (\lambda x.x \notin A) \\
= \; & \forall a \; b. \; a, b \notin A \Rightarrow \\
& \qquad (\lambda x.x \notin (a\,b)\bullet A) = (\lambda x.x \notin A)
\end{aligned}
$$

Quick Calculation

$$
\begin{aligned}
& A \ \mathit{supports}\ (\lambda x. x \notin A) \\
\overset{\text{def}}{=}\ & \forall a\ b.\ a, b \notin A \Rightarrow \\
& \qquad (a\,b) \bullet (\lambda x. x \notin A) = (\lambda x. x \notin A) \\
=\ & \forall a\ b.\ a, b \notin A \Rightarrow \\
& \qquad (\lambda x. (a\,b) \bullet ((a\,b) \bullet x \notin A)) = (\lambda x. x \notin A) \\
=\ & \forall a\ b.\ a, b \notin A \Rightarrow \\
& \qquad (\lambda x. ((a\,b) \bullet (a\,b) \bullet x) \notin (a\,b) \bullet A) = (\lambda x. x \notin A) \\
=\ & \forall a\ b.\ a, b \notin A \Rightarrow \\
& \qquad (\lambda x. x \notin (a\,b) \bullet A) = (\lambda x. x \notin A) \\
=\ & \forall a\ b.\ a, b \notin A \Rightarrow (\lambda x. x \notin A) = (\lambda x. x \notin A)
\end{aligned}
$$

A Slightly More Interesting Datatype

$$
\begin{array}{lcl}
trm & ::= & \mathsf{Var} : name \\
& \mid & \mathsf{App} : trm\ trm \\
& \mid & \mathsf{Lam} : [name].trm \\
& \mid & \mathsf{Let} : [name].trm\ trm \\
& \mid & \mathsf{Letrec} : [name].([name].trm \times trm)
\end{array}
$$

Differences between Let and Letrec:

- $\texttt{let}\ x := e_1\ \texttt{in}\ e_2$ — x is in scope of e_2
- $\texttt{let rec}\ x := e_1\ \texttt{in}\ e_2$ — x is in scope of e_1,e_2

A Slightly More

A simple Ocaml program for factorial:

```
let rec
  fact = fun n -> if n = 0 then 1
                              else n * fact(n - 1)
in
fact 6;;
```

| Let : $[name].trm\ trm$
| Letrec : $[name].([name].trm \times trm)$

Differences between Let and Letrec:

- let $x :=$ e_1 in e_2 — x is in scope of e_2
- let rec $x :=$ e_1 in e_2 — x is in scope of e_1,e_2

A Slightly More Interesting Datatype

$$
\begin{array}{rcl}
trm & ::= & \mathsf{Var} : name \\
& \mid & \mathsf{App} : trm\ trm \\
& \mid & \mathsf{Lam} : [name].trm \\
& \mid & \mathsf{Let} : [name].trm\ trm \\
& \mid & \mathsf{Letrec} : [name].([name].trm \times trm)
\end{array}
$$

Differences between Let and Letrec:

- $\texttt{let}\ x := e_1\ \texttt{in}\ e_2$ — x is in scope of e_2
- $\texttt{let rec}\ x := e_1\ \texttt{in}\ e_2$ — x is in scope of e_1,e_2

Abstractions as Functions

In the datatype-package I am creating, every element of $[name].trm$ is a specific partial function. That means the type of Letrec

$$\mathsf{Letrec} : [name].([name].trm \times trm)$$

will be subset of the function-space

$$name \longrightarrow ((name \longrightarrow (trm)\ option) \times trm)\ option$$

That looks complicated, but it works splendidly.

The Specific Functions

The functions of type

$$name \rightarrow (trm)\ option$$

we are interested in are:

$$\begin{aligned}[a].t \stackrel{\text{def}}{=} \lambda b.\ & \texttt{if}\ b = a\ \texttt{then}\ \mathsf{Some}(t)\ \texttt{else} \\ & \texttt{if}\ b \# t\ \texttt{then}\ \mathsf{Some}((a\,b)\bullet t) \\ & \texttt{else}\ \mathsf{None}\end{aligned}$$

By “Miracle”

…these functions satisfy the properties:

- $\pi \bullet [a].t = [\pi \bullet a].(\pi \bullet t)$
- $[a].t_1 = [b].t_2 \Leftrightarrow$
  $$(a = b \wedge t_1 = t_2)$$
  $$\vee (a \neq b \wedge t_1 = (a\, b) \bullet t_2 \wedge a \# t_2)$$

$\Rightarrow$ relative straight-forward calculations

$$\mathsf{Letrec}([a].([b].t_1, t_2)) = \mathsf{Letrec}([c].([d].t_3, t_4)) \Leftrightarrow$$

$$(a = c \wedge [b].t_1 = [d].t_3 \wedge t_2 = t_4)$$
$$\vee (a \neq c \wedge [b].t_1 = (a\, c) \bullet [d].t_3 \wedge a \# ([d].t_3, t_4))$$

These Functions are Completely Generic

For every atom a of type α and object x of type ι

$$\begin{aligned}[a].x \stackrel{\text{def}}{=} \lambda b.\ & \texttt{if}\ b = a\ \texttt{then}\ \mathsf{Some}(x)\ \texttt{else} \\ & \texttt{if}\ b \# x\ \texttt{then}\ \mathsf{Some}((a\,b)\bullet x) \\ & \texttt{else}\ \mathsf{None}\end{aligned}$$

is a function of type $\alpha \longrightarrow \iota\ option$. They have to:

$$\mathsf{Letrec} : [name].([name].trm \times trm)$$

The only thing we have to know is that α is an atom-type and that over ι an appropriate permutation function is defined (we come to that later).

Abstraction Type

Therefore it makes sense to introduce a type-former

$$[\alpha].\iota$$

for abstraction types. It contains all these specific partial functions. We can show the crucial properties:

$$a \;\#\; [b].x \text{ iff } a = b \vee a \;\#\; x$$

$$\mathsf{supp}([a].x) = \mathsf{supp}(x) - \{a\}$$

provided $\mathsf{supp}(x)$ is finite

α-Equivalence Classes

...still need to be defined by hand

$$
\begin{array}{lcl}
trm & ::= & \mathsf{Var} : name \\
 & \mid & \mathsf{App} : trm\ trm \\
 & \mid & \mathsf{Lam} : name \longrightarrow (trm\ option)
\end{array}
$$

$$
\frac{}{\mathsf{Var}(a) \in \Lambda_\alpha} \qquad \frac{t_1 \in \Lambda_\alpha \quad t_2 \in \Lambda_\alpha}{\mathsf{App}\ t_1\ t_2 \in \Lambda_\alpha}
$$

$$
\frac{t \in \Lambda_\alpha}{\mathsf{Lam}[a].t \in \Lambda_\alpha}
$$

$$
\mathsf{Lam}\ [a].t_1 = \mathsf{Lam}\ [b].t_2 \Leftrightarrow [a].t_1 = [b].t_2
$$

Axiomatic Type-Classes

When we define the type of

$$\text{Letrec} : [name].([name].trm \times trm)$$

as

$$name \rightarrow ((name \rightarrow (trm)\ option) \times trm)\ option$$

we need to ensure that on the product, option and function an appropriate permutation is defined.

```
primrec
```

$$\pi \bullet (x, y) = (\pi \bullet x, \pi \bullet y)$$

Axiomatic Type-Classes

Similarly, in our induction principle

$$\frac{\begin{array}{l}(\forall x\, a)\ P\,(\mathsf{Var}(a))\,x \\ (\forall x\, t_1, t_2)\ ((\forall z)\ P\,t_1\,z) \wedge ((\forall z)\ P\,t_2\,z) \\ \qquad\qquad\qquad\qquad \Rightarrow P\,(\mathsf{App}(t_1, t_2))\,x \\ (\forall x\, a\, t)\ a \# x \wedge\ ((\forall z)\ P\,t\,z) \Rightarrow P\,([a].t)\,x\end{array}}{(\forall x\, t)\ P\,t\,x}$$

we need to ensure that x has finite support (usually instantiated with list, tuples, mixed things etc.).

```
primrec
```

$$\pi\bullet(x, y) = (\pi\bullet x, \pi\bullet y)$$

Axiomatic Type-Classes

```
consts
```
$$perm : name\ prm \Rightarrow \iota \Rightarrow \iota \qquad (\_\bullet\_)$$

`axclass` $ptype \subseteq type$

$$[]\bullet x = x$$
$$(\pi_1 @ \pi_2)\bullet x = \pi_1\bullet(\pi_2\bullet x)$$
$$\pi_1 \sim \pi_2 \Rightarrow \pi_1\bullet x = \pi_2\bullet x$$

```
primrec
```
$$\pi\bullet(x, y) = (\pi\bullet x, \pi\bullet y)$$

Instance Proofs

`primrec`

$$\pi \bullet (x, y) = (\pi \bullet x, \pi \bullet y)$$

`instance` $\times \subseteq (ptype, ptype)\ ptype$

To show:

$$[] \bullet (x, y) = (x, y)$$
$$(\pi_1 @ \pi_2) \bullet (x, y) = \pi_1 \bullet (\pi_2 \bullet (x, y))$$
$$\pi_1 \sim \pi_2 \Rightarrow \pi_1 \bullet (x, y) = \pi_2 \bullet (x, y)$$

Lemmas in this Way

...look as follows:

Lemma: Given $a : name$, $\pi : name\ prm$ and $x : \iota :: ptype$:

$$\pi \bullet [a].x = [\pi \bullet a].(\pi \bullet x)$$

Lemma: Given $a : name$ and $x : \iota :: fstype$:

$$a \ \# \ [a].x$$

Very convenient reasoning, except that it is not general enough...

Only One Type-Parameter

$\texttt{consts}$

$$perm : \alpha \Rightarrow \iota \Rightarrow \iota \qquad (\_\bullet\_)$$

$\texttt{axclass}\ \ ptype \subseteq type$

$$[]\bullet x = x$$

$$(\pi_1 @ \pi_2)\bullet x = \pi_1\bullet(\pi_2\bullet x)$$

$$\pi_1 \sim \pi_2 \Rightarrow \pi_1\bullet x = \pi_2\bullet x$$

Lemma: Given $a : \alpha :: atype$ and $x : \iota :: fstype$:

$$a \;\#\; [a].x$$

Abstractly: Predicates

So I am using very standard predicates:

$$
\begin{aligned}
ptype_{\iota,\alpha} \overset{\text{def}}{=}\\
&(\forall x : \iota.\ []\bullet x = x)\\
\wedge\ &(\forall x : \iota\ \ \pi_1, \pi_2 : \alpha\ prm.\\
&\qquad (\pi_1 @ \pi_2)\bullet x = \pi_1\bullet(\pi_2\bullet x))\\
\wedge\ &(\forall x : \iota\ \ \pi_1, \pi_2 : \alpha\ prm.\\
&\qquad \pi_1 \sim \pi_2 \Rightarrow \pi_1\bullet x = \pi_2\bullet x)
\end{aligned}
$$

Lemma: Given $ptype_{\iota,\alpha}$ and $atype_{\alpha}$:

$$\pi\bullet[a].x = [\pi\bullet a].(\pi\bullet x)$$

Then use Axclasses

Then I am using an axlcass with π_1 and pi_2 of type $name\ prm$:

$$\texttt{axclass}\ \ ptype \subseteq type$$
$$[]\bullet x = x$$
$$(\pi_1 @ \pi_2)\bullet x = \pi_1\bullet(\pi_2\bullet x)$$
$$\pi_1 \sim \pi_2 \Rightarrow \pi_1\bullet x = \pi_2\bullet x$$

and show that

$$ptype_{\iota::ptype,name}$$

holds (I have to do this for every kind of atom the user wants to use — on the SML-level of Isabelle).

Aside

When one has more than one kind of atoms, e.g. name and coname, the properties/predicate:

$$\begin{array}{l}
\mathit{ptype}_{\iota,\alpha} \stackrel{\text{def}}{=} \\
\qquad (\forall x : \iota.\ []\bullet x = x) \\
\quad \wedge\ (\forall x : \iota\ \ \pi_1, \pi_2 : \alpha\ \mathit{prm}. \\
\qquad\qquad (\pi_1 @ \pi_2)\bullet x = \pi_1\bullet(\pi_2\bullet x)) \\
\quad \wedge\ (\forall x : \iota\ \ \pi_1, \pi_2 : \alpha\ \mathit{prm}. \\
\qquad\qquad \pi_1 \sim \pi_2 \Rightarrow \pi_1\bullet x = \pi_2\bullet x)
\end{array}$$

is not enough to characterise permutations.

Aside

When one has more than one kind of atoms, e.g. $name$ and $coname$, the properties/predicate:

> One cannot prove properties such as:
>
> $$\pi_1 \bullet (\pi_2 \bullet x) = \pi_2 \bullet (\pi_1 \bullet x)$$
>
> for $\pi_1 : \alpha_1\ prm$ and $\pi_2 : \alpha_2\ prm$ with $\alpha_1 \neq \alpha_2$.

$$\wedge\ (\forall x : \iota\ \ \pi_1, \pi_2 : \alpha\ prm.$$
$$\pi_1 \sim \pi_2 \Rightarrow \pi_1 \bullet x = \pi_2 \bullet x)$$

is not enough to characterise permutations.

Very Ugly

I need on the abstract level also the predicates:

$$disjoint_{\alpha_1,\alpha_2} \stackrel{\text{def}}{=}$$
$$(\forall \pi : \alpha_1\ prm\ \ x : \alpha_2.\ \pi \bullet x = x)$$
$$\wedge (\forall \pi : \alpha_2\ prm\ \ x : \alpha_1.\ \pi \bullet x = x)$$

and even more ugly:

$$commute_{\iota,\alpha_1,\alpha_2} \stackrel{\text{def}}{=}$$
$$\forall\ \pi_1 : \alpha_1\ prm\ \ \pi_2 : \alpha_2\ prm\ \ x : \iota$$
$$\pi_1 \bullet (\pi_2 \bullet x) = \pi_2 \bullet (\pi_1 \bullet x)$$

Fin

Next week no lecture!