free-from-atom doc[11,40]

The type 'free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
 ->n<-}(T;x;a)' is inhabited (by 'Ax')
iff there exists a token "a" and a term y such that a = "a" in Atom{$n} and x = y in T
such that token "a" does not occur in y.

Thus free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
 ->n<-}(T;x;a)
 is true iff a is an atom and there is some member of the equivalence class of x in T
that is free from a.

To see that this defines a type, we note that if a1 = a2 in Atom{n}, then there is a unique token "a" such that

"a" = a1 = a2 in Atom{n}, and if T1 = T2 in Universe{i} and x1 = x2 in T1,
then any y such that x1 = y in T1 and "a" does not occur in y also satisfies
x2 = y in T2 and "a" does not occur in y.

Thus we justify the rule for equality: freeFromAtomEquality .

One base case is 'free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
 ->n<-}(Atom$n;$a$;$a$)' where $a \in$ Atom$n
. This is not inhabited because every term y = a in Atom$n
must mention the token "a" = a (otherwise we could permute ("a","b") and get y = "b" and hence "b"="a").

Since 'free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}(Atom$n;$a$;$a$)' is not a type unless '$a \in$ Atom$n', if we have
'free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}(Atom$n;$a$;$a$)' as a hypothesis in a sequent
then $a \in$ Atom$n, then since free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}(Atom$n;$a$;$a$)
is not inhabited, the sequent is trivially true.
We thus have the "absurdity rule"" freeFromAtomAbsurdity .

Another base case is that if 'AtomFree(T;x)' then 'Ax $\in x$:$T\|a$'. This is because
AtomFree(T;x) is, by definition,
$\forall a$, b:Atom$n. swap($a$;$b$;$x$) $= x$
, so we may choose b to be "fresh" w.r.t. x (i.e. an atom not occuring in x)
and take y = swap(a;b;x) $= x$
, then whatever token "a" the atom a evaluates to, will not occur in swap(a;b;x).
So, we have the first triviality rule: freeFromAtomTriviality .

The last base case is when x is a closed term not in which token "a" does not occur. Then, as long as
'$x \in T$',
we have, by inspection, 'x:$T\|$"a"'
. Currently, we have to relate the tokens "a" which have parameters of kinds
ut1 or ut2 to the Atom{n} spaces for n=1 or n=2 by explicit matching in the rules, so we need two versions of

this base case rule, one for n=1 and another for n=2. (We are working on a new method for parametrizing the

atom types.) freeFromAtomBase1 freeFromAtomBase2 .

Finally, if 'free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}(A;x;a)' and 'free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.

Successfully scanned:

Not Scanned:
n
->n<-}(u:A→$B(u)$;f;a)' then
then for some token "a", "a" = a in Aton{n}, and there are x' = x in A and
f' = f in u:A→$B(u)$ such that "a" does not occur in f' or x'.
Then $f'(x') = f(x)$ in $B(x)$, and "a" does not occur in $f'(x')$. Therefore,
'free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}($B(x)$;$f(x)$;a)'.
So we have shown that the application rule freeFromAtomApplication is true.

Note that the contrapositive of the application rule in the form
'(¬free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}($B(x)$;$f(x)$;a))
⇒ ((¬free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}(u:A→$B(u)$;f;a)) ∨ (¬free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}(A;x;a)))'
will not be constructively true.
We define 'inheres{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-:n}

$(T;\, x;\, a)$' to be the negation, '¬free-from-atom{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}$(A;x;a)$'
, and we read it as "a is inherent in x:T".
It says that it is not possible to find a representative of x in T which avoids "a", i.e. that

every member of the equivalence class of x in T must mention the atom a.
Now, if $f(x)$
must mention a, there can't be representatives f' and x' of f and x which don't mention a,

so at least one of f or x has no such representative. But since the number of possible representatives is

infinite, we can't in general decide which of them has this property.
So we don't have 'inheres{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-:n}
$(B(x);\, (f(x));\, a)$
⇒ (inheres{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-:n}$((u{:}A\rightarrow B(u));\, f;\, a)$ ∨ inheres{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-:n}$(A;\, x;\, a)$)' in general.

We tried to define inherence as '!condition_cons
inheres{$n:n}
$(T;\, x;\, a)$
$\equiv_{\text{def}}$ $\exists g{:}T\rightarrow \mathbb{B}$. (↑matters{$n:n}$(a;\, g;\, x)$)' where
'matters{$n:n}
$(a;\, g;\, x)$' (read as "matters"(a,g,x))
was a boolean (provided 'atom-free{Error: ScanInteger ->

Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-:n}
(Type; T)') defined by (nu b.
'$\neg_b g(x)$ =b g(swap-atoms{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-}(a;b;x))').
Here, nu b. X[b] means choose a fresh atom b not occring in X and evaluate X[b] to normal form (a boolean in our case

and evalute to that normal form if it does not mention the fresh b and diverge otherwise.

From this definition we could prove (for types that were atom-free) the strong application inherence property

'inheres{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-:n}
($B(x)$; ($f(x)$); a)
$\Rightarrow$ (inheres{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-:n}((u:$A \rightarrow B(u)$); f; a) $\vee$ inheres{Error: ScanInteger ->
Scan Error: Expecting a number.
Successfully scanned:

Not Scanned:
n
->n<-:n}(A; x; a))'
from a purported property of "matters" called
"conservation of matters". Unfortunately, the "conservation of matters" property is not true, as shown by the following

counter-example.
Let g <x,y> = '$\neg_b$(x =a y $\wedge_b$ ($\neg_b x$ =a "a"))',

let f = 'λx.<"a", x>',
let x = "a".
Then g (f x) = g <"a","a"> = 'tt'.
Any tokens "b", "c" different from "a" do not occur in "a",g,f, or x, and
g (f swap(a;b;x)) = g <"a","b"> = 'tt'
g (swap(a;b;f) x) = g <"b","a"> = 'tt'
g (swap(a;b;f) swap(a;c;x)) = g <"b","c"> = 'tt', but
g (swap(a;b;f) swap(a;b;x)) = g <"b","b"> = 'ff'.
This example show that it is possible that
'(↑matters{$n:n}
(a; g; ($f(x)$)))
& (¬(↑matters{$n:n}
(a; ($\lambda X.g(f(X))$); x)))
& (¬(↑matters{$n:n}
(a; ($\lambda F.g(F(x))$); f)))
& (¬(↑matters{$n:n}
(a; (λF.matters{$n:n}($a$; ($\lambda X.g(F(X))$); x)); f)))'
whereas "conservation of matters" purported to show that
'(↑matters{$n:n}
(a; g; ($f(x)$)))
⇒ (((↑matters{$n:n}($a$; ($\lambda X.g(f(X))$); x)) ∨ (↑matters{$n:n}(a; ($\lambda F.g(F(x))$); f)))
∨ (↑matters{$n:n}
(a; (λF.matters{$n:n}($a$; ($\lambda X.g(F(X))$); x)); f)))'