Equality proofs in Cayenne

Lennart Augustsson

Department of Computing Sciences
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Email: augustss@cs.chalmers.se
WWW: http://www.cs.chalmers.se/~augustss

Abstract. We introduce a simple syntactical extension to Cayenne to make equality proofs readable.

1 Introduction

Equality proofs are important for functional programs, since they are encouraged by equational reasoning. Unfortunately, type theory style equational proofs are tedious and look very ugly.

Let us illustrate this with an example. We wish to prove that list append is associative. The function has the following definition:

```
Nil      ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Now we want to establish that `(xs++ys)++zs = xs++(ys++zs)`. With normal notation we do this with an inductive proof[1] as follows.

Base case:

```
  (Nil++ys)++zs     =
= ys++zs            =
= Nil++(ys++zs)
```

by first clause of ++
by first clause of ++, reversed

Inductive step:

```
  ((x:xs)++ys)++zs  =
= (x:(xs++ys))++zs  =
= x:((xs++ys)++zs)  =
= x:(xs++(ys++zs))  =
= (x:xs)++(ys++zs)
```

by second clause of ++
by second clause of ++
by induction hypothesis
by second clause of ++, reversed

Now compare this to the same proof in type theory style. We assume that we have an `Id` type for equality;[2] see appendix A. This proof uses Cayenne/Agda syntax,[Aug98], and assumes there are no hidden arguments.

[1] Assuming finite lists, and no $\perp$.
[2] Which seems to be out of fashion these days.

```
(++) :: (a :: #) -> List a -> List a -> List a
(++) a (Nil) ys = ys
(++) a (x : xs) ys = (:) a x ((++) a xs ys)

appendAssocP :: (a :: #) -> (xs, ys, zs :: List a) ->
        Id (List a) ((++) a ((++) a xs ys) zs) ((++) a xs ((++) a ys zs))
appendAssocP a (Nil) ys zs =
        refl (List a) ((++) a ys zs)
appendAssocP a (x:xs) ys zs =
        subst (List a) (List a)
              ((++) a ((++) a xs ys) zs)
              ((++) a xs ((++) a ys zs))
              (\ (ws :: List a) -> x : ws)
              (appendAssocP a xs ys zs)
```

This is a total mess, and penetrating its meaning is only recommended for masochists. One reason for the mess is all the extra type arguments. As a first step towards readability we allow hidden arguments (as Cayenne does), and use the infix symbol === for the Id type.

```
(++) :: (a :: #) |-> List a -> List a -> List a
(Nil)    ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

appendAssocP :: (a :: #) |-> (xs, ys, zs :: List a) ->
        (xs++ys)++zs === xs++(ys++zs)
appendAssocP (Nil) ys zs =
        refl (ys++zs)
appendAssocP (x:xs) ys zs =
        subst (\ (ws :: List a) -> x : ws) (appendAssocP xs ys zs)
```

The proof we have now is more readable, but it is quite tedious to construct and read. This is not because of any technical problems. This is, in fact, exactly the same proof as presented informally above, but we would be hard pressed see that.

In contrast, the new notational device that we use in Cayenne to make the proof readable will yield the following proof.

```
appendAssocP :: (a :: #) |-> (xs, ys, zs :: List a) ->
        (xs++ys)++zs === xs++(ys++zs)
appendAssocP (Nil) ys zs =
        (Nil ++ ys) ++ zs              ={ DEF }=
        ys ++ zs                       ={ DEF }=
        Nil ++ (ys ++ zs)
appendAssocP (x:xs) ys zs =
        ((x : xs) ++ ys) ++ zs         ={ DEF }=
        (x : (xs ++ ys)) ++ zs         ={ DEF }=
```

```
x : ((xs ++ ys) ++ zs)           ={ appendAssocP xs ys zs }=
x : (xs ++ (ys ++ zs))           ={ DEF }=
(x : xs) ++ (ys ++ zs)
```

The notation is now quite close to the informal proof. The notation is very simple. It is a number of expressions that are equal, separated by the reason why they are equal. If the reason says `DEF` it means that they are equal because of the definitional equations; if it is an expression, then this expression is the proof object that proves that the subexpressions that differ between the two expressions are equal.

2 Equality proof notation

The notation is almost entirely syntactic sugar. There is a simple translation from an equality proof into a normal term. An equality proof has the following form:

exp_1 `={` why_1 `}=`
exp_2 `={` why_2 `}=`
...
exp_n `={` why_n `}=`
exp_{n+1}

The first expression, exp_1, is used to prove that it is equal to itself using reflexivity. Given the proof from an initial segment of an equality chain, it is easy to obtain the next proof. If the reason, why_k, is `DEF` it simply means that the same equality proof can be used even when the next step is added since the type checker can handle definitional equalities. If the reason is a proof object, it is the proof that a subpart of exp_k is equal to a corresponding part of exp_{k+1}. The rest of the two expressions must be identical. Using substitutivity on this proof we can obtain a proof that exp_k is equal to exp_{k+1}. This proof is then combined with the proof from the preceding steps in the chain using transitivity.

The two subexpressions that a given proof object concerns can be explicitly marked by enclosing the subexpression in `[]` (in the first of the two involved expressions). If no subexpression is marked, the type checker will use the smallest subexpressions that make the rest of the expressions identical.

It is almost always sufficent to leave out the `[]` marks and let the smallest changed be derived automatically, but sometimes this is not correct. For example, the expressions "`abs(abs x)`" is equal to "`abs x`", but this is *not* because we have a proof that "`abs x`" is equal to "`x`", so the smallest subexpression that differs is not the right thing to choose in this case.

The extension to the syntax and the translation can be formalized as follows.

The extension to the grammar:

exp ::= "`[`" *exp* "`]`"
exp ::= *eqchain*

eqchain ::= *exp*

eqchain ::= *eqchain* "={" "DEF" "}=" *exp*
eqchain ::= *eqchain* "={" *exp* "}=" *exp*

Note that this is not the concrete grammar used for parsing. An equality proof must have at least one "={ }=" to make sense.

The bracketed expression that indicates what part of the term the equality proof operates on is taken from the expression that precedes the "={ }=".

Example:

```
 f[x]  ={       h  ::   x === y   }=
[f y]  ={       h' :: f y === g y }=
 g[y]  ={ symm h  ::   y === x   }=
 g x
```

The translation:

$$
\begin{aligned}
&\mathcal{E}[\![\, e \,]\!] && = \texttt{refl}\; e \\
&\mathcal{E}[\![\, c = \{\ \texttt{DEF}\ \} = e \,]\!] && = \mathcal{E}[\![\, c \,]\!] :: (h(c) === e) \\
&\mathcal{E}[\![\, c = \{\ p\ \} = C[e] \,]\!] && = \texttt{trans}\; \mathcal{E}[\![\, c \,]\!]\; (\texttt{subst}\; (\backslash\ x\ ->\ C[x])\; p)
\end{aligned}
$$

$$
\begin{aligned}
&h(e) && = e \\
&h(c = \{\ ...\ \} = e) && = h(c)
\end{aligned}
$$

The `DEF` steps can, of course, be left out of an equality proof since they do not contribute to the proof object, but they can increase the clarity of a proof.

3 Possible extensions

3.1 Symmetry

The notation handles substitutivity and transitivity, but not symmetry of equality. It would be easy to extend the type checker to try a given *why* expression with and without symmetry applied, and pick the one that succeeds, but we have not implemented this.

3.2 Proof simplification

The equality proof obtained by the translation described above is not always as simple as it could be. The first time that transitivity is used in a proof we know that the first proof will always be a proof with reflexivity, so it is not really necessary to use transitivity; the second proof to `trans` suffices as the whole proof. Furthermore, sometimes the function argument used to `subst` is the identity function, so there is really no need to use `subst` at all.

It is easy to add these simplifications to the translation process, but it does not really matter since the proof objects are not used anyway.

4 Acknowledgments

Thanks to Jessica for proof reading.

References

[Aug98] Lennart Augustsson. Cayenne — a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP'98)*. ACM Press, September 1998.

A The equality type

```
(===) :: (a :: #) |-> a -> a -> #

refl :: (a :: #) |-> (x :: a) |-> x === x

symm :: (a :: #) |-> (x, y :: a) |->
        x === y -> y === x

trans :: (a :: #) |-> (x, y, z :: a) |->
        x === y -> y === z -> x === z

congr :: (a :: #) |-> (P :: a -> #) -> (x, y :: a) ->
        x === y -> P x -> P y

subst :: (a, b :: #) |->  (x, y :: a) |-> (f :: a -> b) ->
        x === y -> f x === f y
```

B Complete proof of `rev (rev xs) = xs`

```
module example$RevEq =
open System$List use * in
open System$Id use (===), symm in

struct

concrete
(++) :: (a :: #) |-> List a -> List a -> List a
(++) (Nil) ys = ys
(++) (x : xs) ys = x : (xs ++ ys)

appendNilP :: (a :: #) |-> (xs :: List a) ->
        xs ++ Nil === xs
appendNilP |a (Nil) =
        Nil|a ++ Nil                        ={ DEF }=
        Nil
appendNilP |a (x : xs) =
        (x:xs) ++ Nil                       ={ DEF }=
        x:(xs ++ Nil)                       ={ appendNilP xs }=
        x:xs                                ={ DEF }=
        Nil ++ (x:xs)
```

```
appendAssocP :: (a :: #) |-> (xs, ys, zs :: List a) ->
        (xs++ys)++zs === xs++(ys++zs)
appendAssocP |a (Nil) ys zs =
        (Nil|a ++ ys) ++ zs                 ={ DEF }=
        ys ++ zs                            ={ DEF }=
        Nil ++ (ys ++ zs)
appendAssocP |a (x:xs) ys zs =
        ((x : xs) ++ ys) ++ zs              ={ DEF }=
        (x : (xs ++ ys)) ++ zs              ={ DEF }=
        x : ((xs ++ ys) ++ zs)              ={ appendAssocP xs ys zs }=
        x : (xs ++ (ys ++ zs))              ={ DEF }=
        (x : xs) ++ (ys ++ zs)

-----

concrete
rev :: (a :: #) |-> List a -> List a
rev (Nil) = Nil
rev (x : xs) = rev xs ++ (x : Nil)

revAppendP :: (a :: #) |-> (xs, ys :: List a) ->
        rev (xs++ys) === rev ys ++ rev xs
revAppendP |a (Nil) ys =
        rev (Nil|a ++ ys)                   ={ DEF }=
        rev ys                              ={ symm (appendNilP (rev ys)) }=
        rev ys ++ Nil                       ={ DEF }=
        rev ys ++ rev (Nil|a)
revAppendP |a (x:xs) ys =
        rev ((x : xs) ++ ys)                ={ DEF }=
        rev (x : (xs ++ ys))                ={ DEF }=
        rev (xs ++ ys) ++ (x:Nil)           ={ revAppendP xs ys }=
        (rev ys ++ rev xs) ++ (x:Nil)       ={ appendAssocP (rev ys) (rev xs) (x:Nil) }=
        rev ys ++ (rev xs ++ (x:Nil))       ={ DEF }=
        rev ys ++ (rev (x:xs))

revRevP :: (a :: #) |-> (xs :: List a) ->
        rev (rev xs) === xs
revRevP |a (Nil) =
        rev (rev (Nil|a))                   ={ DEF }=
        rev (Nil|a)                         ={ DEF }=
        Nil
revRevP |a (x:xs) =
        rev (rev (x:xs))                    ={ DEF }=
        rev (rev xs ++ (x:Nil))             ={ revAppendP (rev xs) (x:Nil) }=
        rev (x:Nil) ++ rev (rev xs)         ={ revRevP xs }=
        rev (x:Nil) ++ xs                   ={ DEF }=
        (x:Nil) ++ xs                       ={ DEF }=
        x:xs
```