A TENTATIVE PROPOSAL FOR A GENERAL SYNTAX OF HIGHER INDUCTIVE TYPES

1. Overview

First of all, I will consider all inductive types to be specified by elimination rules. Coq implements eliminators in terms of primitive `match` and `fix` operations, whereas conversely it is a theorem that at least for single, ordinary, inductive types, `match` and `fix` can be reduced to eliminators. It is not clear whether there are notions of `match` and `fix` for higher inductive types.

Now, we may say that specifying an inductive definition in terms of an eliminator involves describing three related notions:

- an *algebra* of some sort,
- a corresponding notion of *dependent algebra* over an algebra, and
- a notion of *algebra section* of a dependent algebra.

The resulting inductive type is then an algebra (this gives its constructors) such that any other dependent algebra over it (this gives the hypotheses of the eliminator) there exists a specified algebra section (the eliminator; its being an algebra section specifies the computation rules).

For instance, when describing the natural numbers, we have:

- An algebra is a type X together with an element $z_X : X$ and a function $s_X : X \to X$.
- A dependent algebra over X is a dependent type $X \vdash Y$ together with an element $z_Y : Y(z_X)$ and a function $s_Y : \prod_{x:X}(Y(x) \to Y(s_X(x)))$.
- An algebra section of such a Y is a function $f : \prod_{x:X} Y(x)$ such that $f(z_X) \equiv z_Y$ and $f(s_X(x)) \equiv s_Y(f(x))$.

Similarly, when describing the circle S^1, we might have:

- An algebra is a type X together with $b_X : X$ and a loop $\ell_X : (b_X = b_X)$.
- A dependent algebra over X is a dependent type $X \vdash Y$ together with an element $b_Y : X(b_X)$ and a path $\ell_Y : (\ell_X)_\#(b_Y) = b_Y$.
- An algebra section of such a Y is a function $f : \prod_{x:X} Y(x)$ such that $f(b_X) \equiv b_Y$ and $f_{**}(\ell_X) = \ell_Y$.

Category-theoretically, simple inductive types can be viewed as initial algebras for endofunctors. Suppose we have an endofunctor F which acts on dependent types and sections. That is, for a dependent type $X \vdash Y$, we have a dependent type $FX \vdash FY$, and similarly for a section $f : \prod_{x:X} Y(x)$ we have $F(f) : \prod_{x:FX} FY(x)$. Then we can define:

- An algebra is a type X with a map $a_X : FX \to X$.
- A dependent algebra is a dependent type $X \vdash Y$ together with a term $a_Y : \prod_{x:FX}(FY(x) \to Y(a_X(x)))$. Categorically, we may view this as a

commuting square

(1)
$$\begin{array}{ccc} FY & \longrightarrow & Y \\ \downarrow & & \downarrow \\ FX & \longrightarrow & X. \end{array}$$

- An algebra section of such is a function $f : \prod_{x:X} Y(x)$ such that $f(a_X(x)) \equiv a_Y(Ff(x))$. Categorically, this means that the following square commutes:

(2)
$$\begin{array}{ccc} FY & \longrightarrow & Y \\ \uparrow & & \uparrow \\ FX & \longrightarrow & X. \end{array}$$

For instance, F might be defined by

$$FX \equiv A \times (B \to C \to X). \tag{3}$$

Then we can canonically extend F to act on dependent types by taking $X \vdash Y$ to

$$(a, g) : A \times (B \to C \to X) \ \vdash \ \prod_{b:B, c:C} Y(g(b, c)).$$

and on sections by taking $f : \prod_{x:X} Y(x)$ to

$$\lambda(a, g).\lambda b\, c.f(g(b, c)).$$

The resulting inductive type W will have one constructor of type

$$A \to (B \to C \to W) \to W. \tag{4}$$

In Coq syntax, this would be the inductive type

```
Inductive W : Type :=
| constr : A -> (B -> C -> W) -> W.
```

Inductive types with multiple constructors can be modeled with single endofunctors using coproducts, but syntactically, it is more natural to use multiple endofunctors. (In particular, this is the only way to *obtain* empty types and coproducts as inductive types.) Namely, if we have endofunctors F_i which act on dependent types and sections as above, then we can define an algebra to come with maps $F_i X \to X$ for all i, and so on.

We require the functors F_i to adhere to a special form—strict positivity—so that initial algebras can consistently be expected to exist. The example functor F above is the prototype: we can have constant factors in FX independent of X, and also function types with X as target.

More generally, we allow all function types to be *dependent*, and we allow X to also be dependent on some context Γ of *indices*. To describe a fully general *dependent constructor form*, let Δ be a context, and for each $i = 1, \ldots, n$ let $\Delta \vdash \Theta_i$ be a dependent context, and let $\Delta \vdash \vec{p} : \Gamma$ and $\Delta, \Theta_i \vdash \vec{q}_i : \Gamma$ be vectors of terms. Then the constructor form specified by these data is the following type in the context of a variable $X : \Gamma \to \mathsf{Type}$.

$$\prod_{\vec{x}:\Delta} \left(\left(\prod_{\vec{y}_n : \Theta_n(\vec{x})} X(\vec{q}_n(\vec{x}, \vec{y}_n)) \right) \to \cdots \to \left(\prod_{\vec{y}_1 : \Theta_1(\vec{x})} X(\vec{q}_1(\vec{x}, \vec{y}_1)) \right) \to X(\vec{p}(\vec{x})) \right)$$

This is not exactly an endofunctor F of types in any context. But we can regard F as taking our type X in context Γ and producing a type

$$\left(\prod_{\vec{y}_n:\Theta_n(\vec{x})} X(\vec{q}_n(\vec{x},\vec{y}_n))\right) \times \cdots \times \left(\prod_{\vec{y}_1:\Theta_1(\vec{x})} X(\vec{q}_1(\vec{x},\vec{y}_1))\right)$$

in context $\vec{x} : \Delta$. Then we can say that X is an *algebra* for F if we have a map $FX \to X$ lying over $\vec{p} : \Delta \to \Gamma$. (And, of course, we can have multiple such F's.) There are similar definitions of dependent algebras and algebra sections. For the most part, in the rest of this note, we will speak only about endofunctors of the simple sort, but everything should apply just as well in the fully dependent case with indices.

One final, less commonly mentioned, modification is that instead of considering all the constructors being given at once, we can add them inductively: a specification of an inductive type is then either empty (in which case it specifies the empty type), or such a specification together with an additional "constructor" of a specified sort. This is the point of departure for higher inductive types. Namely, we consider specifications defined inductively like this, but now at each step, we can either add an ordinary constructor as before (a "point-constructor") or a "higher constructor". The goal is to produce notions of "algebra" such as we did for our description of S^1 above. In this case, the inductive ramification is *necessary*, because the types of later constructors (such as $\ell : b = b$ for the circle) must refer to the previous constructors (such as b).

In this note, we will restrict consideration to HITs with only points and 1-paths, no higher paths. Probably most of what I'm going to say could be made to work in the general case, but it would be technically more difficult. Moreover, higher paths are known to not increase the expressiveness of HITs: higher path constructors are (propositionally) reducible to 1-paths using the "hub and spoke" method. (In particular, types of arbitrarily high h-level can be constructed by iterating 1-path constructors parametrized by types with higher homotopy.)

Moreover, no interesting examples of HITs that I know of require k-path constructors for $k > 2$, and not many of them even require $k = 2$. So performing the hub-and-spoke reduction manually when necessary would not be a very big deal. Thus for implementation purposes, I propose to focus on 1-path constructors for now. (It *is* necessary, however, that we allow HITs with more than one 1-path constructor.)

2. A semantic approach

Consider some list of constructors for a higher inductive type. In this section we will suppose, for simplicity, that they are all "simple" in that their domains do not involve dependencies. Thus, a point constructor might look like (4):

$$A \to (B \to C \to W) \to W$$

while a 1-path constructor might look like

$$A \to (B \to C \to W) \to (u = v) \tag{5}$$

where u and v are some terms of type W. We will restrict ourselves to 1-path constructors for simplicity. Note that (5) can be regarded as a homotopy between

two morphisms:

$$FW \underset{v}{\overset{u}{\Downarrow}} W$$

where $FX = A \times (B \to C \to X)$ as in (3).

Now, suppose that we have defined the notions of algebra, dependent algebra, and algebra section for the higher inductive type specified by the first n constructors in our list (where perhaps $n = 0$). Call these *n-algebras*. We now consider what the valid forms are for the $(n+1)^{\text{st}}$ constructor, and what its corresponding $(n+1)$-*algebra* notions should be.

Of course, there are two cases depending on whether the $(n+1)^{\text{st}}$ constructor is a point constructor or a path constructor. If it is a point constructor, then it is specified by the single datum of an endofunctor F (satisfying the same restrictions as for ordinary inductive types). In this case, just like we did in the previous section for a single endofunctor, we define:

- An $(n+1)$-algebra is an n-algebra X together with a map $a_X : FX \to X$.
- A dependent $(n+1)$-algebra over such an X is a dependent n-algebra $X \vdash Y$ together with a term $a_Y : \prod_{x:FX}(FY(x) \to Y(a_X(x)))$.
- An $(n+1)$-algebra section of such a Y is an n-algebra section $f : \prod_{x:X} Y(x)$ such that $f(a_X(x)) \equiv a_Y(Ff(x))$. Note that we ask for a definitional equality here.

Now consider the case of a 1-path constructor. In order to specify the data of such a constructor, we define the auxiliary notion of a *F-point of n-algebras*. This consists of

- In the context of an n-algebra X, a term $s_X : FX \to X$.
- In the context of a dependent n-algebra $Y \twoheadrightarrow X$, a dependent term $s_Y : FY \to Y$, i.e. such that the following square commutes:

$$\begin{array}{ccc} FY & \xrightarrow{s_Y} & Y \\ \downarrow & & \downarrow \\ FX & \xrightarrow[s_X]{} & X. \end{array} \tag{6}$$

- In the context of an n-algebra section $X \to Y$, a homotopy in the following square:

$$\begin{array}{ccc} FY & \xrightarrow{s_Y} & Y \\ \uparrow & \swarrow & \uparrow \\ FX & \xrightarrow[s_X]{} & X. \end{array} \tag{7}$$

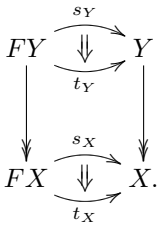
If (7) commutes definitionally, we say that s is a *strict F-point of n-algebras*.

Now, the minimum necessary data for a path constructor consists of an endofunctor F and two F-points of n-algebras, say s and t. Given this, we can define:

- An $(n+1)$-algebra is an n-algebra X together with a homotopy between $s_X : FX \to X$ and $t_X : FX \to X$.
- A dependent $(n+1)$-algebra over such an X is a dependent n-algebra $Y \twoheadrightarrow X$ together with a homotopy between s_Y and t_Y lying over the given

homotopy $s_X \simeq t_X$:

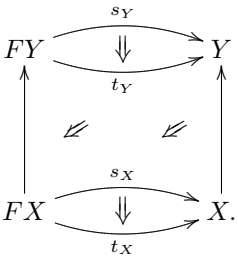
(8)



The front and back squares here commute because s and t are F-points of n-algebras, and $Y \twoheadrightarrow X$ is a dependent n-algebra. (Recall that type-theoretically, the usual way to give a homotopy q lying over a homotopy p is to give q a type such as $p_{\#}x = y$, where $p_{\#}$ denotes transport along p.)

- An $(n+1)$-algebra section of such a Y is an n-algebra section $X \to Y$ together with a higher homotopy filling the following cylinder:

(9)



Here the top and bottom are the homotopies specified in the $(n+1)$-algebra structure of X and the dependent $(n+1)$-algebra structure of Y, and the front and back are the homotopies specified since s and t are F-points of n-algebras and $X \to Y$ is an n-algebra section.

Note that we ask only for a homotopy in the definition of an $(n+1)$-algebra section. This corresponds to asking for a propositional computation rule for the higher constructors of higher inductive types. We do this for various reasons. On the one hand, in most cases, we only know how to construct models that have a propositional rule. On the other hand, in order to have a definitional rule, we'd need to make some arbitrary choices in, e.g., the definition of `map_dep` (versus some doppelganger of it), the definition of "paths over" (do you transport the source or the target, or define a basic inductive type of dependent paths, or...), and so on. It seems unnatural to privilege one such choice by making it the output of a definitional rule.

On the other hand, we *do* want a definitional computation rule for the *point* constructors of an HIT. Getting this would probably be the biggest advantage of a native Coq implementation of HITs (followed by not having to write out the elimination and computation rules by hand). It's a real pain to have to transport along propositional computation rules for points all the time; all the models we know how to build do have definitional computation for point constructors; and there are no arbitrary choices involved. So there is no reason not to make those computation rules definitional.

In the case of dependent functors and inductive families with indices, as described briefly in §1, we might separate out the data slightly differently. We consider the

contexts Δ, $\Delta \vdash \Theta_i$, and the terms $\Delta, \Theta_i \vdash \vec{q}_i : \Gamma$ to be part of the functor F (which takes types in context Γ to types in context Δ, as before). However, at least when describing a 1-path constructor, it is natual to regard the terms $\Delta \vdash \vec{p} : \Gamma$ not as part of F, but as part of an F-point of n-algebras. In particular, the two points u and v can lie over different points of Γ, say $\Delta \vdash \vec{p} : \Gamma$ and $\Delta \vdash \vec{q} : \Gamma$. In this case we would also require, as part of the data, a homotopy in Γ between $\vec{p}$ and $\vec{q}$, and we would ask the homotopy in the definition of an $(n+1)$-algebra to lie over this one in Γ. However, since dependent paths are equivalently paths with transported source, we should be able to restrict to the case when $\vec{p}$ and $\vec{q}$ are the same without real loss of expressivity, and this would probably be easier to implement.

But even with all of that settled, there are several problems with this as a definition. Firstly, there is no syntactic description of what constitutes an F-point of n-algebras. Secondly, if the user had to specify such an F-point explicitly, it would involve giving three terms (s_X, s_Y, and the homotopy), hence *six* terms for each 1-path constructor, instead of the two that, one feels, ought to be enough (its source and target).

And finally, for a sensible notion (in particular, one which exists in models) we expect an F-point to be a natural transformation. The data given above makes it natural up to homotopy, but (unless the point is strict) it seems unlikely that these homotopies need be coherent, which is problematic. For all of these reasons, the definition needs improvement.

3. A syntactic approach

Now let's approach the problem from the syntactic side. Considering the usual informal presentation of a higher inductive type, we may suppose that a 1-path constructor is specified by, in addition to the usual input data F, two terms (its source and target) in the context of the previous constructors and the inputs F.

For example, if we suppose for simplicity that our constructor has the form

$$A \to (B \to X) \to (u = v),$$

then this means that u and v are terms of type X in a context containing a type variable X along with data making X into an n-algebra, a variable $a : A$, and a variable $g : B \to X$.

Obviously, the intent is that when we define a higher inductive type W, the corresponding constructor will be a homotopy between $u_W : FW \to W$ and $v_W : FW \to W$. Here u_W denotes the interpretation of the term u with $X \coloneqq W$ and its n-algebra data, and with the variables a and g abstracted over to produce a function $FW \to W$.

This looks like the first (non-dependent) part of the data for an F-point of n-algebras. However, it is not immediately clear how to derive the rest of the data. In fact, in general it is impossible! Suppose that the ambient context contains a term

$$P : \prod_{X : \mathsf{Type}} (X \to X).$$

Of course, P might be the parametric identity (and if it were defined by a term, then probably it must be so, by parametricity), but in some models there can be other such P's. For instance, classically we could take P to be the nonidentity

automorphism of bool, but the identity on every other type; nothing requires that P be "natural" in any sense.

Now suppose that in the context containing P, we defined an "interval" I with two point-constructors $b_1, b_2 : I$, but instead of $\sigma : b_1 = b_2$ as the path-constructor, we asked for $\sigma : P_I(b_1) = P_I(b_2)$. What would the hypotheses of the dependent eliminator for I say? We should have a dependent type $I \vdash Y$, together with data $y_1 : Y(b_1)$ and $y_2 : Y(b_2)$, but now we should ask for a path in Y living over σ, i.e. an identity $\sigma_\#(?) =?$ in $Y(P(b_2))$. However, nothing ensures that $Y(P(b_1))$ and $Y(P(b_2))$ are even inhabited! We know that $Y(b_1)$ and $Y(b_2)$ are inhabited by y_1 and y_2, but this tells us nothing about $Y(P(b_1))$ and $Y(P(b_2))$.

It is worth noting that the terms $P_I(b_1)$ and $P_I(b_2)$ contain the type I being defined as an argument of a function (namely P). In ordinary inductive types, the only terms which can appear on the "right hand side" of a constructor are *indices*, which are not allowed to contain I as an argument to functions. (Thanks to Kristina for pointing this out.) However, in our case we really do need to allow the terms u and v to contain the type being defined as an argument. For instance, we may want to concatenate paths in u and v, and path-concatenation is a function involving the type as an argument.

In conclusion, we have an example in which there is no way to define a dependent eliminator, and hence no way to extend u and v to F-points of n-algebras. As mentioned in the last section, one solution would be to ask the user to specify, as part of defining a higher inductive type, the other two parts of the data of an F-point. However, we'd like to automate this process to some extent to make things easier on the user, and we'd like to ensure that our homotopies are more likely to be coherent.

With these goals in mind, we will now explain how to derive an entire F-point of n-algebras from *any* term u as above. The catch is that the morphisms $FX \to X$ will not in general be the same as those obtained by naively interpreting u in the appropriate context. However, it seems that in all reasonable cases, they will be at least propositionally equal, and that this might be provable automatically. (In some simple cases, they are even definitionally equal.)

Suppose $FX = A \times (B \to X)$, as above, and that we have a term u given as before. Let Z be the inductive type generated by the first n constructors together with a single additional constructor $g : B \to Z$. Then u can be regarded as defining a term of type Z in the context of a single variable $a : A$.

Note that the elimination rule of Z says that if we have a dependent n-algebra $Z \vdash Y$ together with a term $g' : \prod_{b:B} Y(g(b))$, then there is an n-algebra section $f : \prod_{z:Z} Y(z)$ such that $f(g(z)) \equiv g'(b)$. We are now ready to define our F-point.

- Given an n-algebra X, in the context of $a : A$ and $h : B \to X$, we can apply the non-dependent eliminator of Z to the n-algebra structure of X together with h, to obtain a term $\mathrm{rec}(X, h) : Z \to X$. Then $\mathrm{rec}(X, h)(u(a)) : X$ in context (a, h) gives the desired morphism $u_X : FX \to X$.
- Given a dependent n-algebra $X \vdash Y$, we build $u_X : FX \to X$ as above. What remains is to give, in the context of $a : A$ and $h : B \to X$ and $h' : \prod_{b:B} Y(h(b))$, a term of type $Y(u_X(a, h))$. However, since $\mathrm{rec}(X, h) : Z \to X$ is an n-algebra map[1], the pullback $Z \vdash Y(\mathrm{rec}(X, h)(z))$ is a dependent

[1]We do need to know, for this, that dependent n-algebras pull back along n-algebra maps. This should be provable by induction.

n-algebra, and in the context of $a : A$ and $h : B \to X$ and $h' : \prod_{b:B} Y(h(b))$. Now since $\mathrm{rec}(X, h)(g(b)) \equiv h(b)$, we have $h' : \prod_{b:B} Y(\mathrm{rec}(X, h)(g(b)))$, so we can apply the dependent eliminator of Z to obtain a section $\mathrm{rec}(Y, h') : \prod_{z:Z} Y(\mathrm{rec}(X, h)(z))$. Evaluating this at $u(a)$ yields the desired term $s_Y : FY \to Y$.

- We can obtain (7) by induction on Z.

(It seems at least more likely that these homotopies will be *coherently* natural, since they are produced by induction over an inductive type. However, we have not proven this yet.)

4. Some examples

Now let's consider what this looks like in some examples. First, an example where it gives exactly what we expect. Consider the second constructor for the circle, where the terms u and v are both the basepoint b. The schema above would tell us that to calculate the source of the path ℓ in S^1, we should follow the following recipe. First define an inductive type with one unary constructor (coming from the one previous constructor of S^1, namely b)—that is, a copy of the unit type—then eliminate out of this type into S^1 sending its generating element to the basepoint b. Now evaluate this eliminator at its generating element. Clearly this is merely a roundabout way to specify the basepoint $b : S^1$.

It appears that this definitional coincidence will happen whenever the term u is literally one of the previous constructors of the inductive type, or one of the arguments to the new constructor. This includes many useful examples, such as suspensions, mapping cylinders, homotopy pushouts, and the h-prop truncation.

On the other hand, in bizarre cases the two interpretations may not even be propositionally equal. Consider the counterexample above, with P in context and our "interval" I. Then according to this scheme, the *actual* source and target of the path constructor σ would be obtained by building the two-element inductive type `bool`, applying P to its two points, and then applying the obvious morphism into I. Since P may not be natural, this could be very different from the result of applying P to the two points $b_1, b_2 : I$.

Let's consider a more complicated example: the 0-truncation. The naïvest way to do this is

```
Inductive pi0 (A:Type) : Type :=
| cpnt : A -> pi0 A
| isset : forall (x y : pi0 A) (p q : x = y), p = q.
```

However, this doesn't fit our scheme. We are not generalizing the *input* types of the constructors used for ordinary inductive types, only their *output* types, and `forall (x y : A) (p q : x = y)` is not a valid input type for a constructor of an inductive type. We might consider generalizing further to allow this sort of constructor, but fortunately, we don't really need them. Here is an equivalent definition:

```
Inductive pi0 (A:Type) : Type :=
| cpnt : A -> pi0 A
| isset : forall (l : S1 -> pi0 A), refl (l base) = map l loop.
```

That is, we force all loops in the space `pi0 A` to be contractible. Now the input to the second constructor is of the standard sort we considered above, with $A = 1$ and

$B = S^1$. Finally, we make one more modification, reducing the 2-path constructor to a 1-path constructor with a hub-and-spoke construction:

```
Inductive pi0 (A:Type) : Type :=
| cpnt : A -> pi0 A
| isset_hub : forall (l : S1 -> pi0 A), pi0 A
| isset_spoke : forall (l : S1 -> pi0 A) (x : S1), l x = isset_hub l.
```

Now we again get a definitional equality. The terms u and v are, respectively, `l x` and `isset_hub l`, and since both `l` and `isset_hub` are constructors of the intermediate ordinary inductive type Z, they are preserved definitionally by its eliminators (at least, if we have definitional η-conversion for functions).

Finally, let's consider some examples in which we have only a propositional equality. On the one hand, we might simply be referring to previous path-constructors, which only have a propositional elimination.

```
Inductive sphere : Type :=
| base : sphere
| equator : base = base
| north : equator = refl base
| south : equator = refl base.
```

Rather than go through the work of reducing this to a 1-path constructor, just note that since `equator` is a path constructor, its computation rule is only propositional. Thus the type of, say, `north` (or whatever 1-path constructor replaces it) will not involve `equator` (in X) but rather the image of the corresponding path from the intermediate *higher* inductive type Z.

More than this can happen; consider the projective plane.

```
Inductive RP2 : Type :=
| base : RP2
| equator : base = base
| hemisphere : equator @ equator = refl base.
```

Now the source of `hemisphere` involves a path concatenation, and hence so will the resulting 1-path constructor. But path concatenation is also only preserved propositionally. Clearly, all sorts of operations on paths and higher paths can get into the act here.

As a final example, consider the hub-and-spoke reduction for (say) 2-paths. We start from

```
Inductive stuff :=
| ...
| constr (args) : p = q
| ...
```

with `p` and `q` being parallel 1-path terms. Now we build a version of the 1-sphere generated by two points and two paths between them; then we can write a term `rep p q : S1 -> X` which eliminates these generating paths onto `p` and `q` respectively. The reduction is then

```
Inductive stuff :=
| ...
| constr_hub (args) : stuff
| constr_spoke (args) (x:S1) : rep p q x = constr_hub args
| ...
```

Now aside from `rep p q` having only propositional computation, there is a third issue: lack of η-conversion for the higher inductive `S1`.

5. Potential solutions

Obviously, it will be annoying to the user for the constructors that the system produces to have different types from those asked for. Even more problematically, since one constructor can be used in a later one, if the first one has the "wrong" type, then the "obvious" way to write some later constructor may be ill-typed. Thus, although implementing higher inductive types exactly as described above would be much better than nothing—the user should usually be able to manually transport the constructors, eliminator, and computation rules along the requisite propositional equalities—it would be better to be able to "correct" the constructors.

One possibility would be to stick the user with proof obligations, for each path constructor, to show that its source and target terms, after being modified as in §3, remain propositionally equal to their naive interpretations. Then the named constructor could be defined by concatenating the "real" constructor, obtained as in §3, with the equalities supplied by the user. This would have the advantage that in the case of definitional equality, the system could notice it and *not* require the proof obligation or modify the constructor. One might also hope to automate a search for these equality proofs: in most examples it seems quite obvious what the equality should be, and a well-designed rewrite system might be able to find most of them.

The user would probably also prefer to modify the eliminator and computation rules, so as to refer to the constructors asked for rather than their automatically generated modifications. This is slightly less important, since it could always be done by hand, and at least it doesn't impact the type-checking of later constructors. It's also somewhat trickier, because it would require a "naive interpretation" of the *dependent* part of an F-point, which we have seen does not always exist. So if this were to be given to the user as a proof obligation, it would seemingly require first a *specification* of such a dependent term, and then a proof of its equality with the dependent version of §3, lying over the non-dependent version. (In particular, we would be back to requiring six or even eight data for each path constructor.)

We're hoping that there is a simpler solution. Perhaps a better one will emerge in attempting to implement the approach we have described.