Recursive Types For Free!' (DRAFT)

Philip Wadler

University of Glasgow

July 1990 [some typos fixed Aug 2008]

Recursive types pervade programming: lists, trees, and streams being three of the most common examples. Recursive types come in two principle flavours, least fixpoint or greatest fixpoint.

For example, the least fixpoint least *X*. 1 + *A* × *X* yields lists with elements of type *A*. The least fixpoint least *X*. *A* + *A* × *X* yields binary trees with leaves of type *A*, and the greatest fixpoint greatest *X*. *A* × *X* yields streams with elements of type *A*.

Adding recursive types can alter the nature of a type system. The polymorphic lambda calculus has the pleasant property of being strongly normalising: all reduction sequences terminate in a normal form. But augmenting this calculus with the type least *X*. (*X* → *X*) has the unpleasant consequence of introducing terms with no normal form.

Fortunately, strong normalisation can be preserved by a mild restriction: don't allow the recursive type variable to appear in a negative position. The example violates this constraint, because the recursive type variable *X* appears to the left of the function arrow.

Thus, it is safe to extend the polymorphic lambda calculus by adding least fixpoint types with type variables in positive position. Indeed, no extension is required: such types already exist in the language! If *F* *X* represents a type containing *X* in positive postion only, then least fixpoints may be defined in terms of universal quantification:

This introduces a new type satisfying the isomorphism . Note that it is an isomorphism, not an equality: the type comes equipped with functions and . This formula can be found, for instance, in [Freyd 89] and [Wraith 89]. It is not as widely known as it should be – I know of several computer scientists who have re-invented this particular wheel. The excellent textbook by Girard, Lafont, and Taylor [GLT 89] gives several special cases, but not this general form.

More interestingly, polymorphic lambda calculus also contains greatest fixpoints. These may be defined in terms of existential quantification

again subject to the restriction that *X* appears only positively in *F* *X*. This is a little surprising: greatest fixpoints allow infinite objects, such as streams, yet the strong normalisation property is preserved. Surprising, but not new: it was known previously that greatest fixpoints and strong normalisation could co-exist: both [Hagino 87] and [Mendler 87] describe type systems that include leastand greatestand have strong normalisation. But the encoding provides a simple proof of this fact. Hagino refers to the coding of least fixpoints, but says that he doesn't know of a coding for greatest fixpoints; Mendler doesn't refer to either coding, but gives a lengthy proof that adding least and greatest fixpoints to polymorphic lambda calculus preserves strong normalisation.

(As it turns out, not new either: I have since discovered that Wraith also describes this encoding [Wraith 89], although there is a small technical error: he writes

which is incorrect.)

# Least Fixpoints As Weak Initial Algebras

Let's now look at the fixpoint result in a little more detail. This will require a mild dose of category theory. Don't panic: all terms will be explained as we go along.

By a functor *F* in polymorphic lambda calculus, we will mean an operation taking types into types, and terms into terms, such that if then , preserving identities and composition: and . Every type *A* containing a type variable *X* in positive positions only corresponds to a functor *F* *X* = *A*, which takes a type *T* into the type *F* *B* = *A* [*T*/*X*].

An object in a category is *weakly initial* if there is a map from it to every other object and it is *initial* if this map is unique. In categorical terms, the least fixpoint of *F* is an initial F-*algebra*. An *F*-algebra is a pair (*X*, *f*) consisting of an object *X* and an arrow *f* : *F X* → *X*. These form a category, where a morphism between (*X*, *f*) and (*Y*, *g*) is given by an arrow *h* : *X* → *Y* such that the diagram

f

F X ----------> X

| |

| |

F h | | h

| |

v v

F Y ----------> Y

g

commutes.

Assume least is given by the equation:

As a convenient abbreviation, we will write *T* for least *F*. Then we can define two functions:

It follows immediately that the diagram

In

F T ----------> T

| |

(2) F (fold X k) | | fold X k

V V

F X ----------> X

K

for all *X*:

in

F (least F) -----> least F

| |

(2) F (fold X k) | | fold X k

V V

F X ----------> X

k

commutes. In other words, (*T*, *in*) is an *F*-algebra from which there is a map, called *fold* *X* *k*, to every other *F*-algebra; that is, (*T*, *in*) is a weakly initial *F*-algebra.

Let's consider what this means in a particular case. Take

The values of type have the forms *inl* () and *inr x*, where . If , then is given by

Now, define *Nat* to be the least fixpoint of *F*:

This corresponds to the natural numbers: *in* (*inl* ()) represents zero, and *in* (*inr n*) represents the successor of *n*. Let *k* : 1 + *X* → *X*. Then diagram (2) states that

If we take *z* = *k* (*inl* ()), and *s x* = *k* (*inr x*), then we can rewrite this in the familiar form:

That is, the value of *fold X k* is given for zero (namely, *z*), and is found for (*n* + 1) by recursively applying *fold X k* to *n*, and then applying a given function (namely, *s*) to this result.

As a second example, take

The values of type 1 + *A* × *X* have the forms *inl* () and *inr* (*a*, *x*), where *a* : *A* and x : X. If *f* : *X* → *Y*, then *F f* : 1 + *A* × *X* → 1 + *A* × *Y* is given by

Now, define to be the least fixpoint of *F*:

This corresponds to lists with elements of type *A* : *in* (*inl* ()) represents the empty list, also written *nil*, and *in* (*inr* (*a*, *as*) represents the list constructed with head *a* and tail *as*, also written *cons a as*. Let . Then diagram (2) states that

If we take and , then we can rewrite this in the familiar form

That is, the value of *fold X k* is given for *nil* (namely, *n*), and is found for *cons a as* by recursively applying *fold X k* to *as*, and then using a given function (namely, *c*) to combine *a* with the result.

# Least Fixpoints As Initial Algebras

So far, we have considered only weak initial algebras. We now give necessary and sufficient conditions for this to be a true initial algebra. We first consider the problem for an arbitrary definitions of *T*, *fold*, *in*, and then specialise to the particular definitions we have given.

In order for (*T*, *in*) to be initial, the map *fold X k* must be unique, i.e., the only map that makes diagram (2) commute.

Let h be an arbitrary map from (*X*, *k*) to (*X'*, *k'*); then combining (1) and (2) we have

in

F T ----------> T

| |

| |

F (fold X k) | | fold X k

| |

v k v

F X ----------> X

| |

| |

F h | | h

| |

v v

F X' ---------> X' ,

k'

yielding a map *h* . *fold X k* from (*T*, *in*) to (*X'*, *k'*). But *fold X' k'* should be the only such map! Thus, initiality entails that

k fold X k

F X ----------> X T -----------> X

| | | |

| | | |

(3) F h | | h implies id | | h

| | | |

v v v v

F X' ---------> X' T -----------> X' .

k' fold X' k'

Further, since *fold T in* is a map *T* → *T*, and *id* is also such a map, initiality also implies that

(4) fold T in = id.

Conversely, (3) and (4) imply than (*T*, *in*) is initial.

Choosing an appropriate instance of (3) gives

in fold T in

F T ----------> T T -----------> T

| | | |

| | | |

F h | | h implies id | | h

| | | |

v v v v

F X ----------> X T -----------> X .

k fold X k

The left-hand square states that h is a map from (T,in) into (X,k); and the right-hand square, combined with (4), states that h must be (fold X k). That is, (fold X k) is the only map from (T,in) into (X,k), as required.

If (*T*, *in*) is an initial *F*-algebra, then *in* is an isomorphism. By functoriality, from *in* : *F T* → *T* we have *F in* : *F* (*F T*) → *F T*, hence (*F T*, *F in*) is an F-algebra. The unique map from (*F*, *in*) into this algebra is given by

To see that *in* and *out* are inverses, stare at the following diagram:

in

F T ----------> T

^ ^

| |

F out | | out

| |

| F in |

F (F T) --------> F T

| |

| |

F in | | in

| |

v v

F T ----------> T .

in

The top square is an instance of (2), and the bottom square commutes trivially. Hence *in* ∘ *out* is a map from (*T*, *in*) to (*T*, *in*); but *id* is also such a map, so by uniqueness we have *id* = *in* ∘ *out*. Now from the upper square we have

completing the proof. It is precisely because *in* is an isomorphism that we are justified in calling *T* a fixpoint of *F*, since we have ; it is because *T* is initial that we are justified in calling it a least fixpoint.

The argument in the preceeding two paragraphs works in any category. In our given category, polymorphic lambda calculus, with the given definitions of *fold* and *in*, we can go further. Take (2) as the left-hand square of (3); then the right-hand square becomes,

fold T in

T -----------> T

| |

| |

id | | fold X k

| |

v v

T -----------> X ,

fold X k

or, in symbols,

Reducing both applications of *fold X k* yields

(this is where we use the definition of *fold*). By the eta rules, it follows that

(this is where we use properties of polymorphic lambda calculus). Thus, for the given definitions, (3) implies (4), and hence (*T*, *in*) is an initial *F*-algebra exactly when (3) holds.

Law (3) does not follow from the reduction laws of polymorphic lambda calculus, and indeed there are models that do not satisfy it. But this law is satisfied in many models, including all those having the property of parametricity (see [Reynolds 83], [Freyd 89], [Wadler 89], [Freyd 90], [AMSW 90]). In particular, in the jargon of [Wadler 89], the "Theorem for Free" derived from the type of fold is just this law.

(A technical point: The "Theorems for Free" result really deals with relations, not functions. In the diagram (3) each arrow denotes a relation rather than a function, namely, the relation induced by the function. Except the arrow labeled "id" actually corresponds to the relation id' defined by the relation

Here *r* → *s* relates *f* : *X* → *Y* to *f'* : *X'* → *Y'* if whenever *r* relates *x* to *x'* then *s* relatates *f x* to *f' x'*. And if *G* is an operation on relations and types such that if *r*:*X* ⟷ *X'* then , then ∀ *r*. *G r* relates *g* : ∀ *X*. *G X* to g' : ∀*X'*. *G X'* (these are both the same type), if whenever r is a relation from *X* to *X'* then (*g X*) (*G r*) (*g' X*). It is not the case in all models that *id'* is the identity relation! This is the purpose of the identity lemma in [Reynolds 83] -- in any model satisfying this lemma, *id'* will be the identity. A parametric model is just one that satisfies the identity lemma. Hence, although the "Theorems for Free" result applies in any model, it is only in a parametric model that (3) and (4) must hold.)

# Iterators And Recursors

The operation is what has sometimes been called an iterator. The value of is computed by applying recursively to each substructure of and applying to the result. More formally, we can refer to the substructure of by taking , for some , and we can apply to each substructure of by taking . Then the sentence above can be expressed in symbols:

This fundamental property is just equivalent to saying that is a map of *F*-algebras.

The recursor is a slight variant of the iterator. It is defined by

The value of at is computed by applying recursively to each substructure and applying to the result, and pairing this with the value of . Thus where expects a value computed on the substructure, expects a value computed on the substructure paired with the substructure itself. This is expressed by the laws:

The first is an easy consequence of the definition of . To derive the second, instantiate (3) to yield:

⟨ g, in . F snd ⟩ fold (X × T) ⟨ g, in ∘ F snd⟩

F (X × T) --------> X × T T ----------> X × T

| | | |

| | | |

F snd | | snd implies id | | snd

| | | |

v v v v

F T ----------> T T -----------> T

in fold T in

Then the left square commutes trivially, the top line of the right square is *rec*, and the bottom line is *id* by (4), yielding the desired law.

# Greatest Fixpoints

Greatest fixpoints are exactly dual. We will need a little notation for dealing with existential types: unfortunately, there isn't a standard one. The following typing and reduction rules should serve to introduce the notation, which is based on viewing existential types as generalised products:

### Products:

### Existentials:

We will combine these in a straightforward way, so that, for instance,

will be used as an abbreviation for

Mitchell and Plotkin [MP 88] write as and as .

We now take the dual of the previous development. The greatest fixpoint of a functor *F* coresponds to a terminal *F*-coalgebra. An *F*-coalgebra is a pair (*X*, *k*) consisting of an object *X* and an arrow *k* : *X* → *F X*. And a morphism between (*X*, *k*) and (*X'*, *k'*) is given by an arrow *h* : *X → X'* such that the diagram

k

X ----------> F X

| |

| |

h | | F h

| |

| |

X' ---------> F X'

k'

commutes. Assume the greatest fixpoint is given by the equation:

Then we can define two functions:

It follows immediately that the diagram

k

X ----------> F X

| |

| |

unfold X k | | F (unfold X k)

| |

v v

T ----------> F T

out

commutes. In other words, (*T*, *out*) is an *F*-coalgebra into which there is a map, called *unfold X k*, from every other *F*-coalgebra; that is, (*T*, *out*) is a weakly terminal *F*-coalgebra.

As before, if (*T*, *out*) is truly a terminal *F*-coalgebra, then *out* is an isomorphism, with the inverse given by

Furthermore, we will have that (*T*,s *out*) is terminal iff the condition

k unfold X k

X ----------> F X X -----------> T

| | | |

| | | |

h | | F h implies h | | id

| | | |

v v v v

X' ---------> F X' X' ----------> T

k' unfold X' k'

holds.

(A technical point: this equivalence depends on the equivalence of the surjective pairing rule and the equivalent rule for pair types:

If our calculus contains pairs and existentials as primitives, then it is reasonable to insist on these rules. But if, as is often the case, we define pairs and existentials in terms of universals, then these rules will not necessarily hold. But they will necessarily hold in all models satisfying parametricity! So in parametric models we are assured the existence of true greatest fixpoints.

Example: Streams of integers are yielded by the greatest fixpoint of . The formula above instantiates to

Essentially, what is going on here is that from the abstract type one can only extract terms of the form ), which corresponds to the *i*th element of the stream.

Hagino suggests that streams be defined by the operations *unfold*, *head*, and *tail* [Hagino 87]. We can define these by:

These are just the transposition of unfold and in across the isomorphism

# Pragmatics

Regarding pragmatics, it is well known that the embedding of least fixpoints is less efficient than one would like. For instance, the operation to find the tail of a list takes time proportional to the length of the list: one would hope that this takes constant time. Greatest fixpoints have a dual problem: finding the tail of a stream is cheap, but consing an element onto the front of a stream is more expensive than one would like. So one would still be tempted to add least and greatest fixpoints to a language for pragmatic reasons, though it is good to know that in doing so one does not change the language in any fundamental way (in particular, strong normalisation is still preserved).

Is there a way of coding lists in polymorphic lambda calculus that (a) uses space proportional to the length of the list, (b) performs cons in constant time, and (c) performs tail in constant time? Or is there a proof that this is impossible? So far as I know, this is an open question.

# Acknowledgement

I'm grateful to John Hughes for comments on an earlier version of this text. [Update: And to Fruhwirth Clemens for spotting a typo.]

# Bibliography

[AMSW 90] S. Abramsky, J. Mitchell, A. Scedrov, and P. Wadler, Theorems for free, categorically. In preparation.

[Freyd 89] P. Freyd, Structural polymorphism. Series of three e-mail messages to "types" newsgroup, March 1989.

[Freyd 90] P. Freyd, Recursive types reduced to inductive types. In J. Mitchell, editor, 5'th Symposium on Logic in Computer Science, Philadelphia, June 1990. IEEE.

[GLT 89] J.-Y. Girard, Y. Lafont, and P. Taylor, Types and Proofs. Cambridge University Press, 1989.

[Hagino 87] T. Hagino, A typed lambda calculus with categorical type constructors. In Pitt, et al., editors, Category Theory in Computer Science, Edinburgh, September 1987. LNC 283, Springer Verlag.

[Mendler 87] N. P. Mendler, Recursive types and type constraints in second-order lambda calculus. In 2'nd Symposium on Logic in Computer Science, Ithaca, New York, June 1987. IEEE.

[MP 88] J. C. Mitchell and G. D. Plotkin, Abstract types have existential types. ACM Transactions of Programming Languages, 10(3):470--502, 1988. Preliminary version appeared in Proceedings 12'th ACM Symposium on Principles of Programming Languages, 1985.

[Reynolds 83] J. C. Reynolds, Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, Information Processing 83, pp. 513--523. North-Holland, Amsterdam.

[Wadler 89] P. Wadler, Theorems for free! In 4'th International Conference on Functional Programming Languages and Computer Architecture, London, September 1989. ACM.

[Wraith 89] G. C. Wraith, A note on categorical data types. In Pitt, et al., editors, Category Theory in Computer Science, Manchester, September 1989. LNCS 389, Springer Verlag.