LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Inductively Defined Functions in Functional Programming Languages

Inductively Defined Functions in Functional Programming Languages

by

Rod Burstall

LFCS Report Series

ECS-LFCS-87-25
(also published as CSR-230-87)

April 1987

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

INDUCTIVELY DEFINED FUNCTIONS IN FUNCTIONAL PROGRAMMING LANGUAGES *

R.M. Burstall

Dept. of Computer Science

University of Edinburgh

King's Buildings, Mayfield Road

Edinburgh EH9 3JZ, Scotland, U.K.

Abstract

This paper proposes a notation for defining functions or procedures in such a way that their termination is guaranteed by the scope rules. It uses an extension of **case** expressions. Suggested uses include programming languages and logical languages; an application is also given to the problem of proving inequalities from initial algebra specifications.

1. Introduction

When we define recursive functions on natural numbers we can ensure that they terminate without a special proof by adhering to the schema for primitive recursion. The schema can be extended to other data types. This requires inspection of the function definition to ensure that it conforms to such a schema, involving second-order pattern matching. However if we combine the recursion with a **case** expression which decomposes elements of the data type we can use the ordinary scoping rules for variables to ensure termination, without imposing any special schema. I formulate this proposal for 'inductive' case expressions in ML (Milner 1985) since it offers succinct data type definitions and a convenient **case** expression. Any other functional language with these facilities could do as well.

A number of people have advocated the use of initial algebras to define data types in specification languages, see for example Goguen, Thatcher and Wagner (1978) and Burstall and Goguen (1981). Two aspects of this have worried me somewhat

- we do not have a really convenient way to define functions using the unique homomorphism property of the initial algebra .
- we do not have any obvious way to prove inequations ($a \neq b$) about the data elements from the equations which define the data type.

Inductive case expressions can be seen as a proposal for defining functions by the unique homomorphism property, and I show how we can prove inequations using such functions.

Klaeren has for a long time stressed the use of definitions, as opposed to axioms, to define functions in algebraic specifications. For his treatment see Klaeren (1983).

Apart from their use in programming languages they may be useful as an extension to logical languages, say first-order predicate logic. These

* To appear in Special Issue of JCSS

usually deal exclusively with total functions, so a function definition mechanism which guarantees totality provides a rather simple extension, whereas introducing recursive definitions involves us in the business of partial logics. One might argue that most functions we define are total and when we want a genuinely partial function, say 'eval', we can represent it by a relation.

2. Definitions of data types and functions

A new data type is introduced in ML by giving the alternative ways of constructing elements of that type. Thus for example the natural numbers could be defined by

datatype *nat* = *zero* | *succ* ***of*** *nat*

and lists of natural numbers by

datatype *natlist* = *nil* | *cons* ***of*** (*nat* * *natlist*)

This defines the new type *natlist* together with the constructors

nil: *natlist*
cons: *nat* * *natlist* -> *natlist*

To define functions over such data types we resort to recursion. We use **fun** to define functions, just as *datatype* defines types. Thus

```
fun rec length l = case l of
                      nil          => zero
                    | cons(i, l1)  => 1 + length l1
```

In each case a constructor on the left introduces a number of variables which are bound by matching, for example i and $l1$. Similarly

```
fun rec plus(m, n) = case m of
                        zero     => n
                      | succ m1  => succ(plus(m1, n))
```

We can easily make definitions by recursion which do not terminate. But "obvious" termination is rather common in practical programming, since many functions are defined by primitive recursion.

ML allows **fun** *length l* = ... instead of **fun rec** *length l* = ..., but we want to emphasise the recursion. For non-recursive functions we will drop the **rec**.

3. Defining functions inductively by cases

I would like to propose a variant of the ML **case** construction which makes the termination immediate from the syntax. We will write "**ind case**" for "inductive case". The syntax of an "**ind case**" expression is the same as that of a **case** expression.

Let us call the expression after the word **case** the argument of the **case** expression. Now the new feature for **ind case** is that if a variable v appearing on the left in the matching position inside a constructor has the same type as the argument then not only is v declared for use on the right but so is another special variable named $\$v$. This $\$v$ is bound to the value of the whole **case** expression in which the argument has been replaced by v. The original **ind case** then becomes simply **case**. Some examples will help.

```
fun plus(m, n) = ind case m of
                    zero     => n
                  | succ m1  => succ($m1)
```

Here the new variable $\$m1$ represents the value of the whole **ind case** expression replacing m by $m1$. Thus we could expand to

```
fun plus(m, n) = case m of
                    zero     => n
                  | succ m1  => succ( ind case m1 of
                                        zero    => n
                                      | succ m1 => succ $m1 )
```

Further such expansions will push the **ind case** expression arbitrarily deep in a nested **case** expression and enable us to calculate $plus(m, n)$ for any finite m. By this informal argument we see that since all elements of ML data types are finitely deep **ind case** expressions always terminate. This is the advantage they have over explicit recursion.

We may also note that the $\$m1$ replaces a recursive call of *plus* in the previous definition. We could think of the **ind case** expression in general as standing for some anonymous recursive function applied to the argument expression; the $ sign then corresponds to a recursive call of this function. This recursive call must, by our syntax, be applied to a component of the original argument. Hence the guarantee of termination.

The length example is similarly accomplished without recursion

```
fun length l = ind case l of
                  nil          => zero
                | cons(i, l1)  => succ $l1
```

Another familiar example

```
fun fact n = ind case n of
                zero     => succ zero
              | succ n1  => n * $n1
```

A tree example, summing the numbers on the nodes

```
datatype tree = niltree | node of tree * nat * tree
```

{*niltree* and *node* use the constructor functions, nullary and ternary}

```
fun sum t = ind case t of
               niltree          => 0
             | node(t1, i, t2)  => $t1 + i + $t2
```

The general primitive recursive scheme on natural numbers is

```
fun rec f n = case n of zero => a | succ m => g(m,f m)
```

We can define the functional, *PrimRec*, in the same way with **ind case**

```
PrimRec(a,g)n = ind case n of zero => a | succ m => g(m,$m)
```

So f = *PrimRec*(a,g), and **ind case** will define all the primitive recursive functions. Using higher order functionals, so that g might itself be *PrimRec* we can define a wider class of functions (possibly all provably recursive ones in Peano arithmetic). (I am grateful to John Mitchell for suggesting the definition of *PrimRec*.)

Consider however an alternative definition of *plus*

```
fun rec plus(m, n) =  case m of
                        zero     => n
                      | succ m1  => plus(m1, succ n)
```

Here we apply *plus* recursively to *m1*, but with the parameter *n* increased to *succ n*. There seems to be no way to express such definitions using **ind case**. We can however "curry" the definition of *plus*, and then translate it (noting that in ML **fn** means **lambda**)

```
plus : nat -> (nat->nat)

fun rec plus m = case m of
                   zero     => (fn n. n)
                 | succ m1  => (fn n. plus m1 (succ n))
```

This becomes

```
fun plus m = ind case m of
                   zero     => (fn n. n)
                 | succ m1  => (fn n. $m1 (succ n))
```

Another approach (slightly clunky!) is to add an explicit parameter mechanism

```
ind case m parameter n of
    zero     => n
  | succ m1  => $m1(succ n)
```

You might think of this as syntactic sugar for the second order version. Alternatively we can use assignment. (To do this in ML we need to use a reference variable and use '!' to extract the value from the reference.)

```
fun plus(m,n) = let val sum = ref n in
                    ind case m of
                        zero => !sum
                      | succ m1  => (sum:=!sum+1; $m1)
                end
```

You may regard using assignment as treason to the noble cause of functional programming, but after all this kind of parameterised recursion is a mark of iterative programs and these are written neatly using assignment. You can choose between a second order functional, as in the previous curried version, or the traditional assignment notation. Of course we must not allow references in the data type or circularity might ensue.

The Fibonacci function which recurses on both n-1 and n-2 also presents a problem, but one can overcome this using the ML **'as'** construction which binds a variable to a subpattern.

```
fun fib n = ind case n of
              zero      => succ zero
            | succ zero => succ zero
            | succ(m1 as (succ m2)) => $m1 + $m2
```

Here m1 is a variable bound to the subpattern *succ m2*.

We should say something about mutually recursive data types. ML has a notation for this using **'and'**. For example

```
datatype S = c of T | a
     and T = d of S
```

This corresponds to the signature

```
sorts S,T opns c:T->S, a:S, d:S->T
```

Similarly we can extend the **ind case** notation with **and**'s to separate the components dealing with each of the mutually recursive types. We could use $\$_1$, $\$_2$,... for the results of applying to the various types, but the type of the variable is sufficient to discriminate, so we just use \$. Thus

```
fun rec f s = case s of
                c t1 => succ(g t1)
              | a    => zero
    and g t = case t of
                d s1 => succ(f s1)
```

becomes

```
fun f s = ind case s of
                c t1 => succ $t1
              | a    => zero
              and d s1 => succ $s1
```

Of course one can use second order functionals, like *maplist*, to capture primitive recursion, but they still need termination proofs and programs using them are not very readable.

4. Syntax and semantics of ind case

Let us go over syntax and semantics a little more carefully. Skip to the next section if this bores you.

Data types and signatures

A data type definition is of the general form

$$\begin{aligned}&\textbf{datatype } s_1 = c_{11} \textbf{ of } t_{11} \mid \ldots \mid c_{1n_1} \textbf{ of } t_{1n_1}\\ &\quad \textbf{and } \ldots\\ &\quad \ldots\\ &\quad \textbf{and } s_m = c_{m_1} \textbf{ of } t_{m_1} \mid \ldots \mid c_{mn_m} \textbf{ of } t_{mn_m}\end{aligned}$$

and corresponds to the signature extension

$$\begin{aligned}&\textbf{sorts } s_1,\ldots,s_m\\ &\textbf{opns } c_{11}:t_{11}\text{->}s_1,\ldots,c_{1n_1}:t_{1n_1}\text{->}s_1\\ &\quad \ldots\\ &\quad c_{m1}:t_{m1}\text{->}s_m,\ldots,c_{mn_m}:t_{mn_m}\text{->}s_m\end{aligned}$$

where t_{ij} is a product of some previously defined sorts and the $s_1,\ldots,s_m$. (In fact ML allows the t_{ij} to use '->' as well as the product '*', but we shall not use that here.)
The data type declaration binds the types $s_1,\ldots,s_n$ to the sorts of the initial algebra on this signature and the constructors $c_{11},\ldots,c_{mn_m}$ to the operations of this algebra and to their inverses (when used in a pattern).

Case expressions

Case expressions are of the general form

```
case E0 of
  P1 => E1
| ...
| Pn => En
```

where the patterns P_i are formed from constructors and variables and $E_0,E_1,...,E_n$ are expressions which may use the variables. The patterns are all of the same type as E_0. The expressions $E_0,...,E_n$ are all of the same type and this is the result type of the case expression.

The case expression is evaluated by evaluating E_0 to get a value, say v, and matching v against each pattern in turn until a match is obtained. The corresponding expression, E_i, is then evaluated with this binding for the variables in the pattern. This gives the value of the case expression.

The patterns must be complete in the sense that any value of the appropriate type will match at least one of them.

The syntax of **ind case** is the same as that of **case** (leaving aside the extension to mutual recursion which we have sketched). The semantics is most easily made precise in terms of the 'copy rule'.

First, if no $-variables appear free in the E_i

$$\textit{ind case } E \textit{ of } P_1 \Rightarrow E_1 \mid ... \mid P_n \Rightarrow E_n$$

can be rewritten to the corresponding case statement, deleting **'ind'**. Second, if x is a variable in P_i and

$$I = \textit{ind case } E \textit{ of } P_1 \Rightarrow E_1 \mid ... \mid P_i \Rightarrow E_i \mid ... \mid P_n \Rightarrow E_n$$

then we may rewrite I by replacing $\$x$ in E_i by $I[x/E]$. For example, *even* 1, writing *z* for *zero* and *s* for *succ*, the following expression gives successively

```
ind case s z of z => true | s n => not $n
ind case s z of z => true | s n => not(ind case n of z => true | s n
                                                          => not $n)
    case s z of z => true | s n => not(ind case n of z => true | s n
                                                          => not $n)
not(ind case z of z => true | s n => not $n)
not(ind case z of z => true | s n => not(ind case n of z => true | s n
                                                          => not $n))
not(    case z of z => true | ...)
not true
```

5. Equational data types

The notation used in ML to introduce a recursive data type is just a cute way of defining a signature or more generally a signature extension. The data type is the initial algebra on this signature or more generally the free extension corresponding to this signature extension. In specification languages we may be interested in defining the initial algebra on a signature subject to some equations. Finite strings, bags (alias multisets) and sets are all easily definable by adding equations for identity, associativity, commutativity and absorption. So for specification purposes let us extend the ML syntax slightly to allow equations, introducing a keyword "**under**". Using ___ as an infixed operator for appending, we define strings thus

```
datatype intstring = empty | unit int | intstring ___ intstring
    under   empty ___ s = s
      and   s ___ empty = s
      and   s ___ (t ___ u) = (s ___ t) ___ u
```

The data type is the initial algebra for the signature with these equations (more generally the free extension).

We can define functions recursively on these equational data types, using cases.

For example

```
fun slength s = ind case s of
                    empty  => 0
                | unit i   => 1
                | s1 __ s2 => $s1 + $s2
```

To show that $F(x)$ is well-defined when the type of x is an equational data type satisfying equations $L_1=R_1$, ...,$L_n=R_n$, it is sufficient to show that

$$F(L_1) = F(R_1),\ \ldots,F(L_n) = F(R_n)$$

To see this suppose that $E_1 = E_2$ is provable from these equations, $L_i = R_i$, then there must be a sequence of expressions $E'_1,\ldots,E'_m$ with $E_1 = E'_1$ and $E_2 = E'_m$ such that for each i in $1,\ldots,m$ $E'_i = \sigma_i L_k$ and $E'_{i+1} = \sigma_i R_k$ (or vice versa) for some k in $1,\ldots,n$ and some substitution σ_i. But then $F(E'_1) = F(E'_2) =\ldots= F(E'_m)$, i.e. $F(E_1) = F(E_2)$. Thus F has the same value for any term in the equivalence class generated by the equations, and it is well-defined on the initial algebra.

In our example we have to show that

$$\begin{aligned} &slength(empty \_\_ s) = slength\ s \\ &slength(s \_\_ empty) = slength\ s \\ &slength(s \_\_(t \_\_ s) = slength((s \_\_ t) \_\_ s) \end{aligned}$$

All these follow simply by expanding out the **ind case** statement, not even needing an inductive proof.

6. Proving inequations

From the defining equations it is easy to prove other equations by using the usual properties of equality, substitution, transitivity, etc. But how can we prove inequations? This is less obvious. Do we have to show somehow that a certain equation is <u>not</u> provable from the defining ones?

I want to show how inequations can be proved using another approach. First we note that if there are no equations terms are unequal just if they have different constructors, or (recursively) if they have the same constructor but some pair of components are unequal. This gives us some inequations to start off with e.g. true $\neq$ false, zero $\neq$ succ n.

But what if there are defining equations? We must use the basic property of the initial algebra, the existence of a unique homomorphism to any other algebra which satisfies the equations. Suppose this homomorphism is f. Then we can prove x $\neq$ y by observing that f(x) $\neq$ f(y). Now f(x) and f(y) may take their values in a data type where we already know some inequations. If not we must apply a similar trick to that data type until we get back to a type with no defining equations for which, as we have seen, the inequations are immediate.

The function f, acts as a discriminator, relating the type to another one which is already known. This is of course reminiscent of Guttag's idea of sufficient completeness.

We will use **ind case** to define the initial homomorphism (the connection

is spelled out in the Appendix).

Let us consider bags as an example. Suppose "++" has been declared syntactically to be an infixed operator. We define bags to be unordered sequences, with possible repetitions

datatype *bag* = *empty* | *nat*++*bag*
under *x*++*y*++*b* = *y*++*x*++*b*

It is convenient to write δ_{xy} for ***if*** *x* = *y* ***then*** 1 ***else*** 0

fun *count*(*x*,*b*) = ***ind case*** *b* ***of***
empty => 0
y++*c* => $c + δ_{xy}

To ensure determinacy of this definition we check that

$count(x,y{+}{+}z{+}{+}b) = count(x,z{+}{+}y{+}{+}b)$

that is that count respects the equation for bags.

We will write b_x for $count(x,b)$, as an abbreviation.

Suppose we want to show that $empty \neq x{+}{+}empty$. Since type *nat* has no equations we know that $zero \neq succ\ zero$. But $empty_x = zero$ and $(x{+}{+}empty)_x = succ\ zero$. So $empty \neq x{+}{+}empty$. Note how this depends on the deterministic property of count. Similarly we might show that $x{+}{+}b \neq b$. (My thanks are due to Horst Reichel for help with this example.)

But how do we know that *count* is sufficient to discriminate between all unequal bags? We need to show that different bags have a different count for some *x*. We wish to prove

Theorem $(\forall x.b_x = c_x) \Rightarrow b = c$

For the proof of this theorem we need an auxiliary definition. Assume that "—" has been declared as an infix.

fun *b*—*y* = ***ind case*** *b* ***of***
empty => *empty*
| *x*++*c* => ***if*** *x* = *y* ***then*** *c* ***else*** *x*++$*c*

Thus *b*—*y* deletes one occurrence of *y* from *b* if possible, We need three lemmas for the proof.

Lemma 1. $\forall x.b_x = 0 \Rightarrow b = empty$

Lemma 2. If $b_y > 0$ then $(b—y)_x = b_x - \delta_{xy}$

Lemma 3 If $b_x > 0$ then $x{+}{+}(b—x) = b$

Lemma 1 is immediate, the other two are easily proved by induction.

The proof of the theorem is then by induction on *b*.

For the data type *set* the function analogous to *count* would be membership.

Notice that the initial algebra gives rise to an induction principle and to use this we have to invent a suitable predicate to prove by induction. This comes from the 'no junk' property of the initial algebra. The 'no confusion' property gives rise to inequations, and here to do proofs we have

to invent a suitable discriminant function. There is some pleasant feeling of duality here.

We have made our data definitions in equational logic, but drawn conclusions from them using inequalities and quantifiers. This is an example of the use of two different 'institutions' in one specification language, a trick called 'duplicity' in Burstall and Goguen (1981) and Goguen and Burstall (1984).

The **ind case** notation has some infelicities perhaps in dealing with parameters and mutual recursion. Further ideas for these would be welcome. Its relation with provably total functions and with unique homomorphisms needs further theoretical work.

I hope this is enough to illustrate the technique for proving inequalities and the usefulness of the **ind case** mechanism for defining total functions.

Acknowledgements

I would like to thank Joe Goguen, Horst Reichel, Robin Milner and Andrzej Tarlecki (among others) for illuminating discussions, also the (anonymous) referee for helpful criticism. I am grateful to SERC and BP for support and to Eleanor Kerse for rapid scribing.

References

Burstall, R. and Goguen, J. (1981) An informal introduction to specification using Clear. In Boyer, R. and Moore, J (editor), The Correctness Problem in Computer Science, pages 185–213. Academic Press.

Goguen, J. and Burstall, R. (1984) Introducing institutions. In Logics of Programs. Springer LNCS No. 164, (eds. Clarke and Kozen).

Goguen, J., Thatcher, J. and Wagner, E. (1978) Initial algebra approach to the specification, correctness and implementation of abstract data types. In Current Trends in Programming Methodology, Vol. 4, (ed. R. Yeh) Prentice-Hall, pages 80–149.

Klaeren, H. (1983) Algebraische spezification. Springer.

Milner, R. (1985) The Standard ML core language. Computer Science Dept. Report, Univ. of Edinburgh.

Appendix: Ind case and the unique homomorphism

To see why **ind case** is a way of using initiality to define the unique homomorphism let us start with a very simple example. We assume a type *bool* with an operation *not* and constants *true* and *false*. Now define

```
datatype nat = zero | succ of nat;
val even m   = ind case m of
                 zero     => true
               | succ m1  => not $m1
```

The signature for *nat* corresponding to the data type declaration is

Σ = **sort** *nat* **opns** *zero*:*nat*, *succ*:*nat* -> *nat*

This defines an initial algebra

$A1 = T(\Sigma)$ ($T(\Sigma)$ is the term algebra on Σ.)

The left hand sides of the clauses in the **ind case** correspond to the operations of this signature. The right hand sides define another algebra over it as follows

```
A2 = sort nat    = boolean
     opns zero   = true
          succ b = not b
```

Now there is a unique homomorphism $h:A1 \to A2$ and the value of the **ind case** expression is just $h(m)$, that is this homomorphism applied to the expression after '**case**'. So *even* m = h m and *even* is this unique homomorphism.

More generally the **datatype** declaration uses pre-existing types and corresponds to a signature extension rather than a signature, so we should properly talk of a free extension of the previous algebra rather than an initial algebra.

Of course this is a particularly simple **ind case** expression: the patterns are only one operator deep and all the variables appear with a $. However we can see from this example how to express the definition of a unique homomorphism using **ind case**. We restrict ourselves to a single new sort in Σ (the generalisation to many sorts is obvious enough using a mutually recursive datatype declaration but notationally cumbersome).

Algebra version

Σ = **sorts** s

opns $c_1 : s_{11} \times \ldots \times s_{1m_1} \to s$,

$\ldots$

$c_n : s_{n1} \times \ldots \times s_{nm_n} \to s$

$A1 = T(\Sigma)$ (*or properly* $F(\sigma:\Sigma_0 \hookrightarrow \Sigma)(A_0)$, *the free extension of* A_0)

$A2$ = **sort** $s = S$

opns $c_1(x_{11},\ldots,x_{1m_1}) = E_1$

$\ldots$

$c_n(x_{1n_1},\ldots,x_{nm_n}) = E_n$

$h:A1 \to A2$ *is the unique homomorphism.*

Functional language version

$$\textbf{datatype } s = c_1 \textbf{ of } s_{11} *\ldots* s_{1m_1} |\ldots| c_n \textbf{ of } s_{n1} *\ldots* s_{nm}$$

$$\begin{aligned}\textbf{fun } h\ x = \textbf{ind case } x \textbf{ of}\\ c_1(x_{11},\ldots,x_{1m_1}) &= E_1[\$x_{11}/x_{11},\ldots,\$x_{1m_1}/x_{1m_1}]\\ |\ \ldots\\ |\ c_n(x_{n1},\ldots,x_{nm_n}) &= E_n[\$x_{n1}/x_{11},\ldots,\$x_{nm_n}/x_{nm_n}]\end{aligned}$$

Let us look at more complicated **ind case** expressions, first one with a free variable, n. We have to parameterise the target algebra and the homomorphism by n.

Functional language version

$$\begin{aligned}\textit{fun } plus(m,n) = \textbf{ind case } m \textbf{ of}\\ zero &=> n\\ |\ succ\ m1 &=> succ(\$m1)\end{aligned}$$

Algebra version

Σ as before, $A1$ as before

$$\begin{aligned}A3_n = \textbf{sort } nat &= nat\\ \textbf{opns } zero &= n\\ succ\ k &= succ\ k\end{aligned}$$

Note that the right hand sides of these definitions are to be interpreted in the algebra of natural numbers, $A1$.
$h_n:A1 \rightarrow A3_n$ is the unique homomorphism $plus(m,n) = h_n(m)$

Now consider the treatment of patterns more than one deep.

Functional language version

$$\begin{aligned}\textbf{fun } f\ n = \textbf{ind case } n \textbf{ of}\\ zero &=> zero\\ |\ succ\ zero &=> zero\\ |\ succ(succ\ m) &=> succ\ \$m\end{aligned}$$

We need to introduce a derived signature with operators corresponding to $zero$, $succ\ zero$ and $\lambda m.succ(succ\ m)$.

Algebra version

Σ as before, $A1 = T(\Sigma)$ as before

$$\Sigma' = \textbf{sorts } nat \textbf{ opns } z{:}nat,\ sz{:}nat,\ ss{:}nat \rightarrow nat$$

Since we insisted on a complete set of patterns there is a total function $r:|A1| \rightarrow |A1'|$ representing each Σ-term as a Σ'-term; this is defined by the matching, recursively, and will depend on the sequence of the patterns unless they are disjoint.

$$\begin{aligned}A2' = \textbf{sorts } nat &= nat\\ \textbf{opns } z &= zero\\ sz &= zero\\ ss\ k &= succ\ k\end{aligned}$$

$h':A1' \rightarrow A2'$ is the unique homomorphism

$f\ n = h'(r\ n)$, the homomorphism applied to the representation of n.

Now consider a pattern variable being used without a \$ as well as with a \$.

Functional language version

```
fun fact n = ind case n of
                zero  => succ zero
              | succ m => (succ m)*$m
```

We have to retain the value of m as well as that of $\$m$, so we use a target algebra which supplies both these values.

Algebra version

Σ as before, $A1 = T(\Sigma)$ as before

```
A2 = sorts nat = nat × nat
     opns zero = (zero, succ zero)
          succ(m,k) = (succ m, (succ m)*k)
```

h:$A1$ -> $A2$ is the unique homomorphism (so $h\ n = (n,n!)$)
$fact\ n = second(h\ n)$

Summary

We have shown how to translate initial algebra definitions of unique homomorphisms to **ind case** expressions. In the opposite direction we have shown how a number of forms of **ind case** expression can be viewed as unique homomorphism definitions, but we have not given a general translation scheme; the examples make it at least plausible that such a scheme could be devised. The problem is open for a formal demonstration.