INDUCTIVELY DEFINED FUNCTIONS

(Extended Abstract)

R.M. Burstall
Dept. of Computer Science
University of Edinburgh
King's Buildings, Mayfield Road
Edinburgh EH9 3JZ
Scotland, U.K.

1. Introduction

A number of people have advocated the use of initial algebras to define data types in specification languages, see for example 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 about the data elements.

I sketch here a proposal for defining functions by the unique homomorphism property, and I show how we can prove inequations using such functions. The function definition mechanism can also be seen as a programming language proposal for "inductive" case expressions and I formulate it in ML syntax (Milner 1984).

2. Definitions in ML

A new data type is introduced in ML by giving the alternative ways of constructing elements of that type. Thus for example to introduce (linear) lists of integers

```
type rec intlist = nil | cons of (int * intlist)
```

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

```
nil: intlist
cons: int * intlist -> intlist
```

The natural numbers could be defined by

```
type rec nat = zero | succ of nat
```

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

```
val 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

```
val 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.

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.

```
val 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

```
val plus(m, n) = case m of
                   zero    . n
                   succ m1 . succ( ind case m1 of z
                                     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

```
val length l = ind case l of
                 nil         . zero
                 cons(i, l1) . succ $l1
```

Another familiar example

```
val fact n = ind case n of
               zero    . succ zero
               succ n1 . n * $n1
```

A tree example, summing the integers on the nodes

```
type rec tree = niltree | node of tree * int * tree

val sum t = ind case t of
              niltree          . 0
              node(t1, i, t2) . $t1 + i + $t2
```

Consider however an alternative definition of *plus*

```
val 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 **fun** means lambda)

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

val rec plus m n = case m of
                       zero   . (fun n. n)
                       succ m1 . (fun n. plus m1 (succ n))
```

This becomes

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

This is not particularly nice but the best I can do. Any better ideas?

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.

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. Equational data types

The notation used in ML to introduce a recursive data type is just a cute way of defining a signature. The data type is the initial algebra on this signature. 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

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

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

For example

```
val slength s = ind case s of
                    empty . 0
                    unit i . 1
                    s1 ___ s2 . $s1 + $s2
```

But in order for this definition to be deterministic we have to check some equations derived from the ones for strings

```
      0 + n = n
      n + 0 = 0
l + (m + n) = (l + m) + n
```

All these are elementary properties of +.

To derive the equations to be checked one may note that the right hand sides define operations corresponding to *empty*, *unit* and ___ and we have to show that these operations obey the same equations as the constructors. (I am still a little fuzzy about a good way to say this precisely.) These new operations are the operations of the target algebra of the unique homomorphism which is being defined by the **ind case** expression.

5. 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 not 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 ≠ false, zero ≠ 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 ≠ y by observing that f(x) ≠ 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 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.

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

type rec *bag* = *empty* | *int*++*bag*
under *x*++*y*++*b* = *y*++*x*++*b*

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

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

To ensure determinacy of this definition we check that $\lambda(y,\$c).\$c + \delta_{xy}$ satisfies the equation for ++, that is

$(\$c + \delta_{xy}) + \delta_{xz} = (\$c + \delta_{xz}) + \delta_{xy}$

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

Suppose we want to show that *empty* ≠ *x*++*empty*. Since type *nat* has no equations we know that *zero* ≠ *succ zero*. But $empty_x$ = *zero* and $(x{+}{+}empty)_x$ = *succ zero*. So *empty* ≠ *x*++*empty*. Note how this depends on the deterministic property of count. Similarly we might show that *x*++*b* ≠ *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.

val *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 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 deadline for this invited paper arrived before I had fully understood even these rather elementary matters. Please forgive the sketchy and tentative nature of this contribution.

Acknowledgements

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

References

Burstall, R. and Goguen, J. 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, 1981.

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

Milner, R. The Standard ML core language. Computer Science Dept. Report, Univ. of Edinburgh, 1984.