Induce-Statements and Induce-Expressions: Constructs for Inductive Programming

Theodore S. Norvell

Department of Computer Science
University of Toronto
norvell@cs.toronto.edu

Abstract. A for-loop is somewhat similar to an inductive argument. Just as the truth of a proposition $P(n+1)$ depends on the truth of $P(n)$, the correctness of iteration $n+1$ of a for-loop depends on iteration n having been completed correctly.
This paper presents the induce-construct, a new programming construct based on the form of inductive arguments. It is more expressive than the for-loop yet less expressive than the while-loop. Like the for-loop, it is always terminating. Unlike the for-loop, it allows the convenient and concise expression of many algorithms. The for-loop traverses a set of consecutive natural numbers, the induce-construct generalizes to other data types.
The induce-construct is presented in two forms, one for imperative languages and one for functional languages. The expressive power of languages in which this is the only recursion construct is greater than primitive recursion, namely it is the multiply recursive functions in the first order case and the set of functions expressible in Gödel's system T in the general case.

0 Data Types

We consider languages in which some of the data types are defined by recursion as in Hoare's 'Recursive Data Types' [8] or the language ML. The example in this paper use the following (polymorphic) types.

$$\begin{aligned}
\mathsf{tree_2}(\alpha) &= \mathsf{empty} \mid \mathsf{node}(\alpha, \mathsf{tree_2}(\alpha), \mathsf{tree_2}(\alpha)) \\
\mathsf{list}(\alpha) &= \mathsf{nil} \mid \alpha.\mathsf{list}(\alpha) \\
\mathsf{tree_n}(\alpha) &= \mathsf{empty} \mid \mathsf{node}(\alpha, \mathsf{list}(\mathsf{tree_n}(\alpha))) \\
\mathsf{natural} &= 0 \mid \mathsf{natural}'
\end{aligned}$$

(with the usual abbreviations $1 = 0', 2 = 1' = 0'', ...$). Files may be considered as lists of characters, or as lists of lines, each line being a list of characters.

The members of a data type are all the values that can be finitely generated from its constructors. It is not possible to generate the same member in more than one way.

A programming methodology for functional programming with such recursive data types is presented by Burstall [1] which is largely the inspiration for the present paper. The main progress that this paper makes over that one, is to present an inductive programming construct to match the inductive data types and to extend the ideas into the realm of imperative programming.

1 Informal Description and Examples

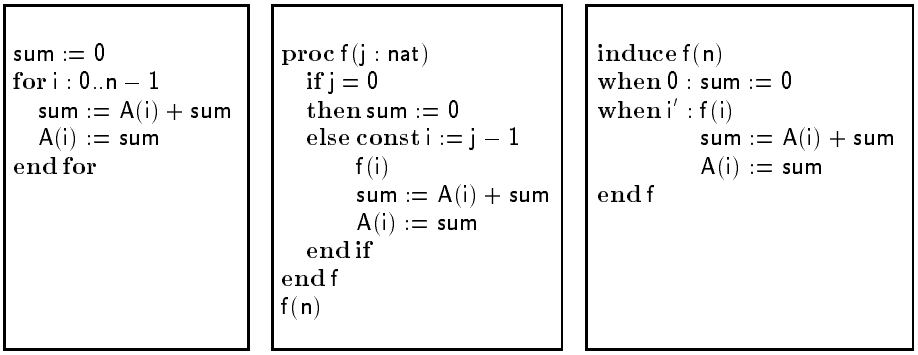

Fig. 0. (a) (b) (c)

1.0 An example. Consider the statement in Fig. 0(a). Using a procedure definition this algorithm might be written as in Fig. 0(b). In this second form the inductive structure is clearer, though somewhat buried in clutter, most of which accompanies the parameter j. The then-part of the if-statement represents the base case, while the else-part represents the induction step. In this particular case a for-loop can be used just as well, but in general, one may want an algorithm where the recursive call occurs elsewhere or where there are multiple recursive calls.

To provide a structured way to write such statements, we introduce two new statement forms, the *induce-statement* and the *repeat-statement*. Before describing their syntax and semantics precisely, we give some examples. The for-loop above can be rewritten as the induce-statement in Fig. 0(c).[0] This works as follows. Assume that the value of n is 1. The value 1 ($= 0'$) is matched against each of two patterns 0 and i'. It matches the second pattern, the pattern variable i (which is local to the statement sequence that follows) is instantiated to the value 0, and the three-statement sequence following the second pattern is commenced. The first statement, f(i), is an example of a repeat-statement. Its execution causes the entire process to repeat except with the value of i (which is 0) rather than the value of n. This time the first pattern matches, so sum is initialized to 0. The repeat-statement, f(i), is now complete so the following two statements are executed and after that the entire induce-statement is then complete.

The induce-statement can be seen as both declaring and calling a local procedure and the repeat-statement as recursively calling this procedure. In this respect the induce-statement is similar to the "label" special form of some LISP dialects [12].

1.1 The Restriction guaranteeing termination. There is an important restriction on induce-statements, the purpose of which is to guarantee that they

[0] The particular syntax used in this presentation is intended to be consistent in spirit with Euclid and its progeny [9].

can not be the source of nontermination: The argument of a repeat-statement must be one of the variables in the pattern that guards it.

```
induce in_order_traversal(T)
when empty :
when node(label, left, right) :
   in_order_traversal(left)
   visit(label)
   in_order_traversal(right)
end in_order_traversal
```

```
const Letter := init("000", "111", "abc", "def",
                     "ghi", "jkl", "mno", "prs",
                     "tuv", "wxy")
var Word : array 0..6 of character
induce columns(7)
when 0 : print Word
when d' : induce letters(3)
          when 0 :
          when i' : Word(d) := Letter(Phone(d), i)
                    columns(d)
                    letters(i)
          end letters
end columns
```

Fig. 1. (a) (b)

1.2 Some More examples. The previous example showed the induce-statement only as a poor alternative to the for-statement. But the repeat-statement need not occur at the beginning of the statement sequence following its pattern. By placing it at the end we obtain a "count-down" for-loop. By placing it in the middle we can express algorithms that would otherwise require a recursive procedure. Further, there can be more than one repeat-statement after a pattern. The idea generalizes easily to any directly recursive data type. These last two points are illustrated by Fig. 1(a).

Induce-statements may be nested, of course. In some cases it may be desirable to repeat an outer loop from within an inner loop. This gives rise to mutual recursion. For example, Fig. 1(b) is a "power loop" [11] that prints all seven letter words that, on a North American phone, correspond to a seven-digit telephone number stored in an array Phone.

2 More Formal Description

2.0 Syntax. Here is the syntax used for induce-statements in this paper

$$
\begin{array}{lll}
\mathit{Statement} & \to & \textbf{induce}\ \mathit{Name}(\mathit{Arg}) \\
 & & \mathit{Clauses}\ \textbf{end}\ \mathit{Name} \\
 & | & \mathit{Name}(\mathit{Variable}) \\
 & | & \mathit{Statement}\ \mathit{Statement} \\
 & | & \mathit{etc.} \\
\mathit{Clauses} & \to & \textbf{when}\ \mathit{Pattern} : \mathit{Statement} \\
 & | & \mathit{Clauses}\ \mathit{Clauses}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Pattern} & \to & \mathit{constructor} \\
 & | & \mathit{constructor}(\mathit{VarList}) \\
 & | & \mathit{Variable}' \\
 & | & \mathit{Variable}.\mathit{Variable} \\
\mathit{VarList} & \to & \mathit{Variable} \\
 & | & \mathit{VarList}, \mathit{VarList} \\
\mathit{Name} & \to & \mathit{identifier} \\
\mathit{Variable} & \to & \mathit{identifier} \\
\mathit{Arg} & \to & \mathit{Expression}
\end{array}
$$

2.1 Context constraints. First, two definitions that will be used in the context constraints (rules of static semantics).

- The *argument type* of an induce-statement is the type of its argument.
- A repeat-statement is *associated* with the smallest enclosing induce-statement with the same identifier.

The following context constraints apply:

0. The patterns in the when-clauses of an induce-statement must correspond one-to-one with the cases of the type definition for the argument type of the induce-statement.
1. The variables in the pattern are local to the following statement and may not be assigned. The type of each a variable can be inferred from its position within the variable list, and from the constructor.
2. Every repeat-statement must have an associated induce-statement.
3. The variable in a repeat-statement must be of the same type as the argument type of the associated induce-statement.
4. The variable in a repeat-statement must occur in the pattern of the when-clause —of the associated induce-statement— that contains the repeat statement.

2.2 Semantics. We consider only the predicative semantics [6, 7] of the induce-statement. The approach would be similar if a different semantic formalism were used, e.g., weakest preconditions or denotational semantics.

In predicative semantics each statement is considered to be a predicate relating the values of variables in the initial and final states. We denote the value of a variable v in the initial state by v and the value of that variable in the final state by $\acute{v}$.

Let **when** $c(v_0, v_1, \ldots, v_m) : S$ be a clause in an induce-statement with expression E. It is considered to be the predicate

$$\langle \exists v_0, v_1, \ldots, v_m \cdot E = c(v_0, v_1, \ldots, v_m) \wedge S \rangle$$

provided none of the v_i are free in E. A case-statement

$$\textbf{case}\ E \quad W_0 \quad W_1 \quad \ldots \quad W_n\ \textbf{end case}$$

is considered to be $\langle \exists i : 0, 1, \ldots, n \cdot W_i \rangle$.

Now the statement **induce** $f(n)$ Z **end** f is considered to be $f(n)$ where f is the (unique) solution of

$$f = \lambda p \cdot \textbf{case}\ p\ Z\ \textbf{end case}$$

That gives the semantics of induce-statements in an untimed model of computation. In a timed model there is a distinguished state variable t representing the current time. In the *recursive time* model [6, 7] this variable is of type $\mathsf{nat} \cup \{\infty\}$, and the time variable is used only to count recursive calls, thus providing an approximation to within a constant factor of the real time. To find the interpretation of **induce** $f(n)$ Z **end** f we first construct $\hat{Z}$ in which every repeat-statement S (refering to f) in Z is replaced by the sequence $t := t + 1\ S$. The induce-statement is considered to be $f(n)$ where f is the (unique) solution of

$$f = \lambda p \cdot \textbf{case}\ p\ \hat{Z}\ \textbf{end case}$$

3 Extensions

```
induce hanoi(7, from := "a",
                to := "b",
                other := "c")
when 0 :
when i' :
  hanoi(i, from, other, to)
  print "Move from ", from, " to ", to
  hanoi(i, other, to, from)
end hanoi
```

(a)

```
induce traverse_tree(T)
when empty :
when node(label, children) :
  visit(label)
  induce traverse_tree_list(children)
  when nil :
  when child.siblings :
    traverse_tree(child)
    traverse_tree_list(siblings)
  end traverse_tree_list
end traverse_tree
```

(b)

Fig. 2.

3.0 Parameters. We can improve the idea by adding a combined parameter and argument list to the induce-statement and an argument list to the repeat-statement. For example, the Hanoi program becomes Fig. 2(a). The changes to the syntax and semantics are straightforward.

3.1 Mutually Recursive Types. As mentioned above mutual recursion between nested loops is permitted. However, the context constraints above prevent some very natural mutual recursions such as this prefix traversal of an n-ary tree shown in Fig. 2(b). Notice that the repeat statement **traverse_tree**(**child**) violates a constraint because **child** was not declared in the current pattern of the **traverse_tree** induce-statement.

To allow this sort of algorithm the constraint on the variable of the repeat statement is loosened. A variable v is *connected* to an induce-statement S iff

- v is in a pattern of S, or
- v is connected to an induce-statement T and the argument for that statement is a variable connected to S

Context constraint (4) on repeat-statements becomes: The variable of each repeat-statement must be connected to the induce-statement associated with the repeat-statement.

3.2 Functional Programming. A similar construct, the induce-expression, is possible for functional programming. For example, the following expression reverses a list, L:

```
induce reverse(L, accumulator := nil)
when nil : accumulator
when head.tail : reverse(tail, head.accumulator)
end reverse
```

The syntax, context conditions, and semantics are straightforward.

4 Programming Methodology

A *refinement calculus* is a triple $(\mathcal{S}, \mathcal{P}, \sqsupseteq)$ in which $\sqsupseteq$ is a transitive and reflexive relation on a set of specifications $\mathcal{S}$, and $\mathcal{P}$ —a set of programs— is a subset of $\mathcal{S}$. Design and coding from a specification S can be seen as the process of finding a program P such that $S \sqsupseteq P$. A refinement calculus for functional programs is discussed by Norvell and Hehner [15] and is used here.

Let **assert** G **in** P be the same as specification P except that G may be assumed to be true. That is, any Q that refines P in circumstances where G holds, refines **assert** G **in** P. For induce statements (or expressions) with natural argument type the following refinement rule holds for any variable n.

$$
\begin{array}{l}
P(n) \sqsupseteq \textbf{induce } f(n) \\
\qquad \textbf{when } 0 : \textbf{assert } n = 0 \textbf{ in } P(n) \\
\qquad \textbf{when } m' : \textbf{assert } m' = n \wedge (P(m) \sqsupseteq f(m)) \textbf{ in } P(n) \\
\qquad \textbf{end } f
\end{array}
$$

The rule is called *refinement by induce*. Of course a similar rule holds for each argument type. Another important rule is that the induce-construct is monotonic with respect to refinement. Other refinement calculi will have corresponding rules.

We will look at an example of deriving an algorithm to search a binary search tree. We will use the type $\mathsf{tree\_2}(\alpha)$ for some linearly ordered type α. We define predicates $\in$ and st

$$
\begin{array}{rl}
b \in \mathsf{empty} \equiv & \textbf{false} \qquad (0) \\
b \in \mathsf{node}(a,u,v) \equiv & (b = a \vee b \in u \vee b \in v) \qquad (1) \\
st(\mathsf{empty}) \equiv & \textbf{true} \qquad (2) \\
st(\mathsf{node}(a,u,v)) \equiv & \quad st(u) \wedge st(v) \qquad (3) \\
& \wedge \langle \forall b : \alpha \cdot (b \geq a \Rightarrow \neg\, b \in u) \wedge (b \leq a \Rightarrow \neg\, b \in v) \rangle
\end{array}
$$

The problem to solve is $P(t)$ where $P(t) =$ **assert** $st(t)$ **in** $b \in t$. That is we require an expression to say whether an element b is in a search tree t. The derivation of such an expression is shown in Fig. 3. Each box represents a subderivation, that is a derivation starting with the expression just above it. In subderivations we can make use of any information provided by the context, in particular from assertions and if-conditions. The two hints "induction hyp." refer to the assertions $P(u) \sqsupseteq f(u)$ and $P(v) \sqsupseteq f(v)$ respectively.

5 Proof by Programming

The laws and techniques of refinement calculi can be turned to an unexpected and interesting purpose. Take $\mathcal{S}$ to be a set of predicates, $\mathcal{P}$ to be $\{\textbf{true}\}$ and interpret $S \sqsupseteq T$ by $\langle \forall \sigma \cdot S \Leftarrow T \rangle$ (where σ is all free variables of S and T) as is done in predicative programming [6, 7]. Then if we can derive $S \sqsupseteq \textbf{true}$, that derivation is a proof of S. Expressing proof as program derivations leads to a way of presenting proofs similar to that of Wim Feijen.

Programming constructs such as if-statements and let-statements have interpretations as predicate-constructions and can be used to structure proofs. Of

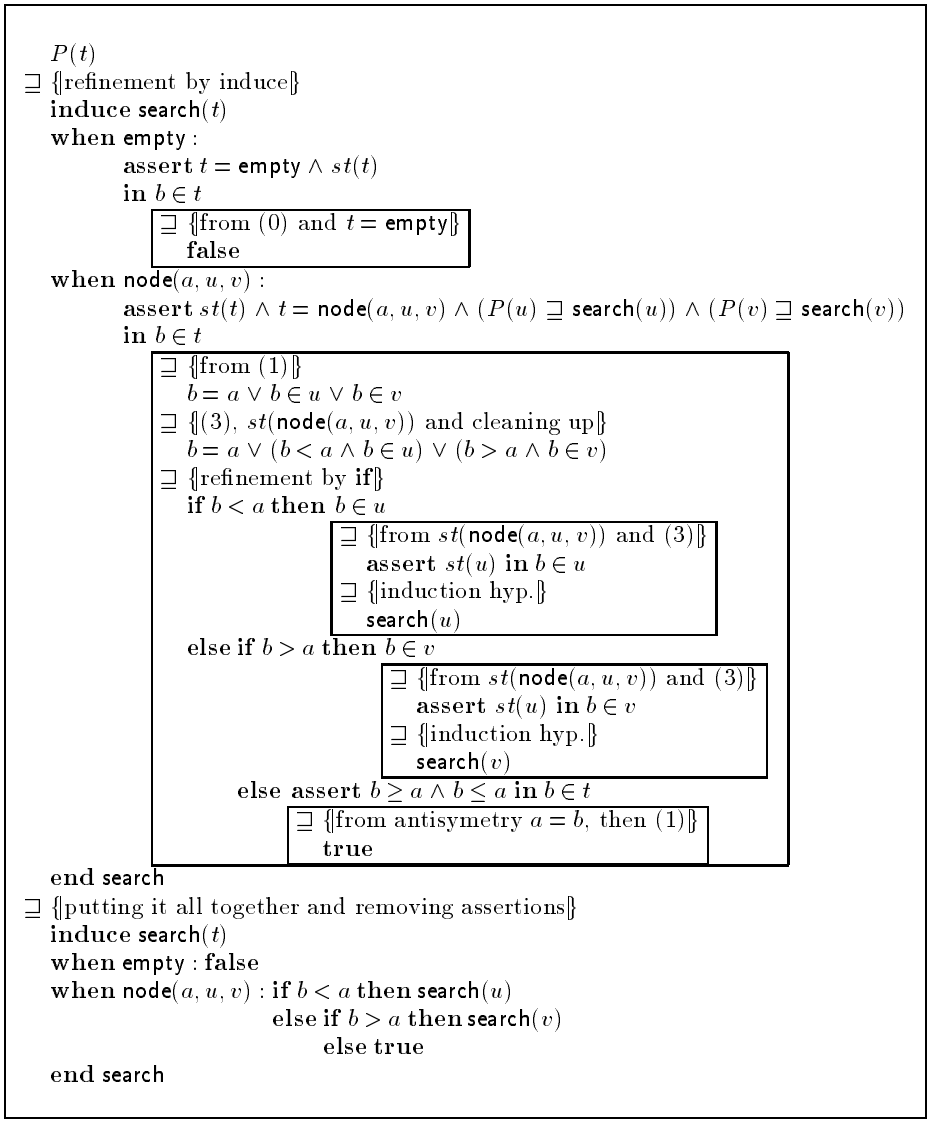

Fig. 3. A derivation of a search program

course induce-statements are used to express inductive proofs. In addition to refinement by induce, we need the following *idempotence* rule

$$\textbf{induce } f(n) \textbf{ when } 0 : P \textbf{ when } m' : f(m) \quad \sqsupseteq \quad P$$

Fig. 4 shows a proof of a simple theorem from number theory.

The advantage of this proof format is that it clearly shows the structure of the argument and allows it to be presented in a way that can be read at a variety of

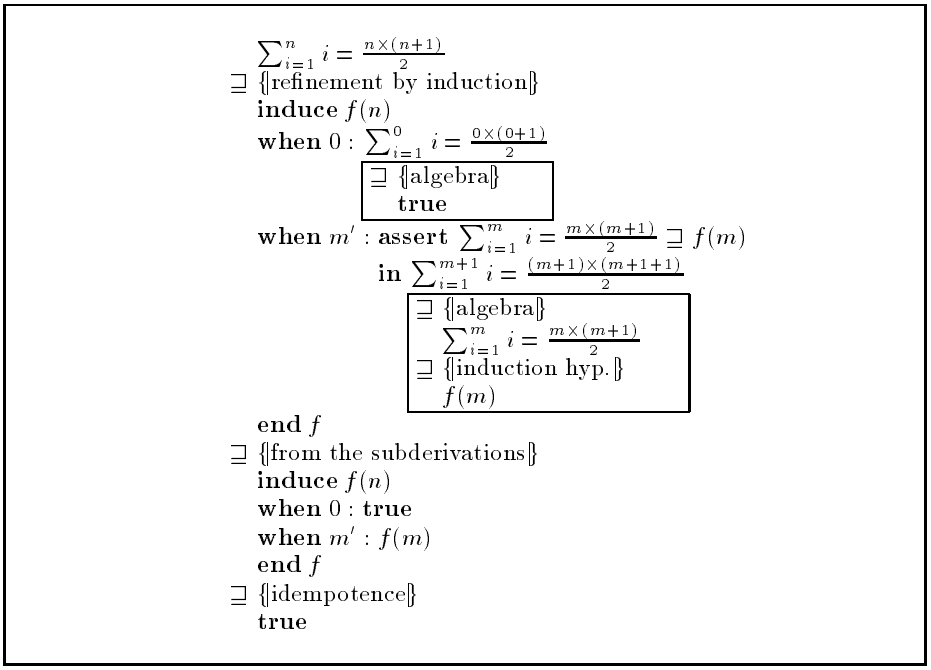

Fig. 4. Proof of $\sum_{i=1}^{n} i = \frac{n\times(n+1)}{2}$

levels of detail. Arguments based on transitivity of refinement and monotonicity with respect to refinement can be easily verified visually.

This way of looking at proofs has the slogan "proof by programming". It should not be confused with constructive type theory where programs are proofs; here it is the process of programming —or rather the record of that process— that is the proof. Proof by programming makes exact the parallel between a circular argument and a nonterminating program. It is important to the soundness of this process that no nonterminating programs occur in the derivation; in this application, the boundedness of induce-expressions is of fundamental importance.

6 Expressive Power

A *total language* is a language in which it is possible only to express total functions, or in other words, terminating computations. Total languages are useful because they support the usual laws of mathematics. For a simple example: $f(n) \times 0$ is the same as 0. A more advanced example is found in the parametricity theorem, which gives "theorems for free," and which can not be proved for languages containing general recursion; for this reason Wadler suggests exploration of practical languages that restrict recursion [19]. The induce-statement provides a possible basis for total languages.

The most commonly known total language is that of primitive recursive arithmetic. A classic example of a total computable function that is not primitive recursive is due to Péter (usually called Ackermann's function) [16, 17] on the

left.

$$\begin{aligned} \mathsf{p}(\mathsf{0},\mathsf{n}) &= \mathsf{n}' \\ \mathsf{p}(\mathsf{m}',\mathsf{0}) &= \mathsf{p}(\mathsf{m},\mathsf{1}) \\ \mathsf{p}(\mathsf{m}',\mathsf{n}') &= \mathsf{p}(\mathsf{m},\mathsf{p}(\mathsf{m}',\mathsf{n})) \end{aligned}$$

$$\begin{aligned} &\textbf{induce } \mathsf{p}(\mathsf{m},\mathsf{n} := \mathsf{n}) \\ &\textbf{when } \mathsf{0} : \mathsf{n}' \\ &\textbf{when } \mathsf{m}' : \textbf{induce } \mathsf{q}(\mathsf{n}) \\ &\qquad\qquad \textbf{when } \mathsf{0} : \mathsf{p}(\mathsf{m},\mathsf{1}) \\ &\qquad\qquad \textbf{when } \mathsf{n}' : \mathsf{p}(\mathsf{m},\mathsf{q}(\mathsf{n})) \\ &\qquad\qquad \textbf{end } \mathsf{q} \\ &\textbf{end } \mathsf{p} \end{aligned}$$

Yet the expression on the right computes Ackermann's function at (m,n). On the other hand, a simple diagonalization argument shows that no total language can express all total computable functions.

Next we look at the exact expressive power for a first-order version and for an unrestricted version of a language based in induce-expressions.

6.0 First-order subset. The first-order subset is called FOI; its syntax is presented in Fig. 5. It has only natural variables, zero, successor, function calls (including repeat-expressions), nonrecursive function definition, and induce expressions. The context conditions given in sections 2.1 and 3.1 are in effect. Any expression in FOI, with free variables of type *nat* only, defines a function from tuples of naturals to naturals. Any function that can be expressed in this language is called FO *inductive*. Any function expressible in this language using induce-expressions nested at most k deep is said to be k FO *inductive*.

Types for FOI and I:
$T \to \mathsf{nat} \mid T \times \cdots \times T \mid T \to T$

The language FOI :

$$\begin{aligned} E \to\ & V_{\mathsf{nat}} \mid 0 \mid E' \\ \mid\ & V_{\mathsf{nat}\times\cdots\times\mathsf{nat}\to\mathsf{nat}}(E,\ldots,E) \\ \mid\ & \textbf{let } V_{\mathsf{nat}\times\cdots\times\mathsf{nat}\to\mathsf{nat}} \\ & \quad (V_{\mathsf{nat}},\ldots,V_{\mathsf{nat}}) = E \textbf{ in } E \\ \mid\ & \textbf{induce } V_{\mathsf{nat}\times\mathsf{nat}\times\cdots\times\mathsf{nat}\to\mathsf{nat}} \\ & \quad (E, V_{\mathsf{nat}} := E,\ldots, V_{\mathsf{nat}} := E) \\ & \textbf{when } 0 : E \\ & \textbf{when } {V_{\mathsf{nat}}}' : E \\ & \textbf{end } V_{\mathsf{nat}\times\mathsf{nat}\times\cdots\times\mathsf{nat}\to\mathsf{nat}} \end{aligned}$$

The language I :

$$\begin{aligned} & E_{\mathsf{nat}} \to 0 \mid {E_{\mathsf{nat}}}' \\ & E_{\alpha\times\cdots\times\beta} \to (E_\alpha,\ldots,E_\beta) \\ & E_{\alpha\times\cdots\times\beta\to\gamma} \to \lambda V_\alpha,\ldots,V_\beta \cdot E_\gamma \\ & E_\alpha \to V_\alpha \mid E_{\beta\to\alpha}\, E_\beta \\ & \quad \mid \textbf{induce } V_{\mathsf{nat}\times\beta\times\cdots\times\gamma\to\alpha} \\ & \qquad (E_{\mathsf{nat}}, V_\beta := E_\beta,\ldots, V_\gamma := E_\gamma) \\ & \qquad \textbf{when } 0 : E_\alpha \\ & \qquad \textbf{when } {V_{\mathsf{nat}}}' : E_\alpha \\ & \qquad \textbf{end } V_{\mathsf{nat}\times\beta\times\cdots\times\gamma\to\alpha} \end{aligned}$$

With α, β, and γ ranging over all types. Any nonterminal V_α represents identifiers of type α.

Fig. 5. The languages FOI and I.

The first-order language is of particular importance because it corresponds most closely to imperative languages. Even in the realm of functional languages, the expressiveness of the first-order subset is important as the excessive use of higher-order functions can lead to unclear programs as surely as their excessive nonuse.

For k : **nat**, a *k-fold recursive definition* is a function definition with the following restriction. Calls to previously defined functions are allowed, but calls

to the function being defined are allowed only if one of the first k parameters is reduced by 1, and all the parameters to the left of that one are unchanged [16, 17]. k-fold recursive definition can not be a source of nontotality as each recursive call comes closer (in lexicographic order) to the base case. A *k-fold recursive function* is one that can be defined using k-fold recursive definition starting with zero and the successor function as basis constants.

Any 1-recursive function is primitive recursive and vice versa. Ackermann's function is an example of a 2-fold recursive function,

Theorem 1. *Any k-fold recursive function is k FO inductive.*

Theorem 2. *Any function which is k FO inductive is $(2k-1)$-fold recursive.*

Both these theorems can be proved by presenting and proving a translation from one form of definition to the other.

The set of all functions that are k-fold recursive for some k, is called the multiply recursive functions. And so FOI expresses exactly the multiply recursive functions. Péter [17] gives two other characterizations to this same class of functions: those functions expressible with transfinite recursion of type ω^k for some natural k; or, those functions expressible using primitive recursion on second order functions.

6.1 The full language. We extend FOI by allowing typed lambda expressions with functional argument and result types. We also allow induce-expressions to be of functional type. Let-expressions are dropped as they can be simulated with lambda.[1] This new language we call I; it is shown in Fig. 5. A function from a tuple of naturals to the naturals expressible in I is said to be *inductive*.

Gödel's system T is a language just like I except that it does not have induce-expressions, and it does have a combinator R_α for each type α,

$$E_{\alpha\times(\alpha\times\mathsf{nat}\to\alpha)\times\mathsf{nat}\to\alpha} \quad\to\quad R_\alpha$$

with the properties

$$R_\alpha(a,f,0) = a$$
$$R_\alpha(a,f,n') = f(R_\alpha(a,f,n),n)$$

Terms of T are easily translated to terms of I. Each occurrence of R is just replaced by a simple induce-expression. Translating terms of I to terms of T is only slightly harder. First any induce-expression with a parameter/argument list can be replaced by an application of an induce-expression without a parameter/argument list. Second any induce expression

induce $f(n)$
when $0 : G$
when $m' : F[m, f(m)]$
end f

can be translated to $R(G,(\lambda fm, m \cdot F[m,fm]),n)$ where fm is a new name.

[1] In fact let-expressions can also be simulated by induce-expressions and hence were not needed in FOI. They were included there so that unnecessary nesting of induce expressions may be avoided.

Theorem 3. *The functions expressible in T are the functions expressible in I.*

Here are two other characterizations of the same class of functions: those functions that can be proven total using Peano arithmetic [5]; or the union of all Grzegorczyk classes $\mathcal{E}^n$ for ordinal $n < \epsilon_0$ [18].

7 Other Work

A similar expression form has been conceived independently by Burstall [2, 3] as an extension to ML. His proposal differs from the one presented here in three respects:

0. It applies only to functional languages.
1. the counterparts of induce expressions are not named and so mutual recursion is not easily expressed.
2. It does not have the equivalent of our parameters.

Parameters can be simulated with higher-order functions, but this is not an option in imperative languages. Likewise mutual recursion of the sort illustrated by the Péter function or the phone number example can only be expressed using higher-order functions. Burstall does give an extension of his notation that can handle the mutual recursion of the n-ary tree example without higher-order programming. In that case the two sorts of repeat-statements can be distinguished by the type of the argument. Distinguishing them by name is more general and permits better type checking.

Malton [10] annotates every loop with an integer expression denoting the maximum number of times it can iterate. This allows expression of exactly the first-order functions expressible in system T. The **charity** language based is on two varieties of data types: "initial" types, whose values are finite, and "final" types, whose values are infinite [4]. Values of the final types are constructed lazily and can not provide a source of nontermination; for example, there is no way to write a function to return the last element of an infinite list because there is no empty list of the right type and hence no way to express the concept of "the last element". The control structures based on initial types are similar to primitive recursion. Meertens proposes a programming construct called "paramorphism" [13], similar to Gödel's R combinator. The Girard/Reynolds calculus has been proposed as a programming language [0]. Data can be represented using Church encodings and hence serve as control structures themselves. The expressive power of the Girard/Reynolds calculus exceeds system T. Type theory has also been proposed as a programming language [14]. The expressive power of some type theories exceed even the Girard/Reynolds calculus.

Acknowledgements Thanks are due to Ray Blaak, Rod Burstall, Ian Hayes, Ric Hehner, and Natarajan Shankar for discussions and comments on earlier versions of this paper.

References

0. Val Breazu-Tannen and Albert R. Meyer. Computable values can be classical. In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 238–243, 1987.
1. R.M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12:41–48, 1969.
2. R.M. Burstall. Inductively defined functions. In *Mathematical Foundations of Software Development*, pages 92–96. Number 185 in Lecture Notes in Computer Science, Springer Verlag, 1985.
3. R.M. Burstall. Inductively defined functions in functional programming languages. *Journal of Computer and System Sciences*, 34:409–421, 1987.
4. Robin Cockett and Tom Fukushima. About CHARITY. Unpublished draft, 1992.
5. Jean-Yves Girard. *Proof Theory and Logical Complexity, Vol. 1.* Number 1 in Studies in Proof Theory. Bibliopolis, 1987.
6. Eric C.R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14:133–158, 1990.
7. Eric C.R. Hehner. *A Practical Theory of Programming.* Springer-Verlag, 1993.
8. C.A.R. Hoare. Recursive data types. *International Journal of Computer and Information Science*, 4:105–132, 1975.
9. Richard C. Holt, Philip A. Matthews, J. Allan Rosselet, and James R. Cordy. *The Turing Programming Language: Design and Definition.* Prentice Hall, 1988.
10. Andrew Malton. *Functional Interpretation of Programming Methods.* PhD thesis, University of Toronto, 1990.
11. Robert Mandl. On "PowerLoop" constructs in programming languages. *SIGPLAN Notices*, 25(4):73–82, 1990.
12. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Communications of the ACM*, 3(4):184–195, 1963.
13. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
14. Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory.* Clarendon Press, 1990.
15. Theodore S. Norvell and Eric C.R. Hehner. Logical specifications for functional programs. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Mathematics of Program Construction*, number 669 in LNCS, pages 269–290. Springer Verlag, 1993.
16. Rósa Péter. Konstruktion nichtrekursiver Funktionen. *Mathematishe Annalen*, 111:42–60, 1935.
17. Rósa Péter. *Recursive Functions.* Academic Press, 1967.
18. H. E. Rose. *Subrecursion: Functions and Hierarchies.* Number 9 in Oxford Logic Guides. Clarendon Press, 1984.
19. Philip Wadler. Theorems for free! In *The Fourth International Conference on Functional Programming Languages and Computer Architecture.* ACM Press, 1989.

This article was processed using the LaTeX macro package with LLNCS style