The Principle of Structural Induction

Dilian Gurov
KTH Royal Institute of Technology, Stockholm

August 25, 2016

We explain the principle of structural induction, targeting mainly undergraduate students, and in particular the students of the two courses DD1350 Logic for Computer Science and DD1361 Programming Paradigms given at KTH.

1 Datatypes as Term Sets

We present the principle of structural induction in the context of sets of terms. This means that we assume a set of function symbols with given arities (i.e., numbers of arguments). Terms are formed from variables and function symbols (constants can be viewed as 0-ary function symbols).

Inductively Defined Datatypes Here we shall be concerned with inductively defined sets of terms, or **datatypes**. Such datatypes are very common in programming languages, and in Computer Science in general. An example of such a datatype are **integer lists**, which can be defined by a BNF grammar with two rules, as follows:

$$\langle \mathsf{IntList} \rangle ::= \mathsf{empty} \mid \mathsf{cons}(\langle \mathsf{Int} \rangle, \langle \mathsf{IntList} \rangle)$$

Here, the datatype IntList is defined in terms of the datatype Int, the definition of which is omitted, and recursively of itself. The function symbols empty and cons are the so-called **constructors** of the datatype, of arity 0 and 2, respectively.

Intuitively, the definition says that an integer list is either empty, or else is a **construction**, consisting of an integer and a integer list; the first argument defines the **head** of the list, while the second argument defines

its **tail**. The rules are only allowed to be applied a finite number of times, hence all lists are finite. Three examples for integer lists are:

$$\mathsf{empty} \quad \mathsf{cons}(11, \mathsf{empty}) \quad \mathsf{cons}(-7, \mathsf{cons}(11, \mathsf{empty}))$$

The concrete syntax for such lists varies from one programming language to another. For instance, in PROLOG these three lists would be written as:

$$[] \quad [11|[]] \quad [-7|[11|[]]]$$

or, formatted for better readability, as:

$$[] \quad [11] \quad [-7, 11]$$

Immediate Subterms The principle of structural induction is based on the **immediate subterm relation** between terms of a given datatype (i.e., of one term being an immediate subterm of another term from the same datatype). An important observation is that BNF grammars make this relation explicit.

For instance, the above BNF grammar guarantees that every integer list l:

- either matches empty, and then has no (immediate) sublists, or else
- matches $\mathsf{cons}(k, l')$ for some integer k and integer list l', and then l' is the one and only immediate sublist of l.

Observe that, in general, every formation rule of a BNF grammar leads to exactly one case of matching that uniquely determines the immediate subterms; we thus have two cases for integer lists. Notice also that each such **destruction** of terms (i.e., "backward" application of rules) is guaranteed to terminate in a finite number of steps, inevitably reaching the atomic (i.e., 0-ary) constructors of the datatype; for integer lists this is the empty list. In other words, we say that the immediate subterm relation is **well-founded**.

For example, the list $\mathsf{cons}(-7, \mathsf{cons}(11, \mathsf{empty}))$ matches the second rule, with its tail $\mathsf{cons}(11, \mathsf{empty})$ as its only immediate sublist. The tail also matches the second rule, with its tail empty as its only immediate sublist. The latter matches the first rule and has no immediate sublists.

2 Definition and Proof by Structural Induction

The principle of structural induction can be used both for defining functions over inductively defined datatypes and for proving properties over such datatypes.

Definition by Structural Induction Structural induction prescribes the definition of functions over all terms of a given datatype by **reducing** the definition to the same function, but over the immediate subterms (of the same datatype) of the term. As we stressed above, we shall use the fact that BNF grammars make the immediate subterm relation explicit. And as the relation is well-founded for such inductive definitions, the principle guarantees that such functions are **well-defined**, in the sense that they define a value for every element of the datatype!

Let's say we want to define formally the notion of list length for all integer lists. For this we introduce a unary function symbol *length*. To follow the principle of structural induction (for datatypes defined inductively through BNF grammars) would mean that we have to make two defining clauses that use a variable l to range over arbitrary integer lists:

- for the case $l = \mathsf{empty}$, we have to express $\mathit{length}(l)$ directly, without referring to *length* again, as empty has no immediate sublists;
- for the case $l = \mathsf{cons}(k, l')$ for some integer k and integer list l', we are allowed to **use** $\mathit{length}(l')$ in the definition of $\mathit{length}(l)$.

In other words, we have to reduce the definition of length of a (non-empty) integer list to the length of the tail of the list. For instance, the definition:

$$\begin{array}{rcl} \mathit{length}(\mathsf{empty}) & = & 0 \\ \mathit{length}(\mathsf{cons}(k, l')) & = & \mathit{length}(l') + 1 \end{array}$$

follows the principle. In PROLOG, the definition of list length would use a predicate, and would look as follows:

```
length([], 0).
length([_ | T], N) :- length(T, NT), N is NT + 1.
```

Note that the definition need not be correct just because we followed the principle (for example, we could have defined the length of the empty list as 1 and then the definition would not capture the intended notion), but the principle guarantees that the function is well-defined **for all** integer lists. So, we can compute the length of the list $\mathsf{cons}(-7, \mathsf{cons}(11, \mathsf{empty}))$ as follows, by applying the defining clauses:

$$\begin{array}{rl} & \mathit{length}(\mathsf{cons}(-7, \mathsf{cons}(11, \mathsf{empty}))) \\ = & \mathit{length}(\mathsf{cons}(11, \mathsf{empty})) + 1 \\ = & \mathit{length}(\mathsf{empty}) + 1 + 1 \\ = & 0 + 1 + 1 \\ = & 2 \end{array}$$

always reducing the computation to the tail of the "current" list. Again, as the sublist relation is well-founded, the computation is guaranteed to terminate for any integer list.

In summary, we must have **one defining clause per formation rule** of the BNF grammar, and reduction is always to the immediate subterms uniquely determined by the formation rule at hand.

Proof by Structural Induction The same way of reasoning can also be applied when we want to prove that some property holds for all terms of a datatype: to follow the principle of structural induction means to reduce the proof of the property for terms to the (proof of the) same property, but for the immediate subterms (of the same datatype) of the term.

So, to prove some property P over all integer lists by structural induction means that, for the arbitrary integer list l, the proof must again consider two cases:

- for the case $l = \mathsf{empty}$, we have to prove $P(l)$ directly, as empty has no immediate sublists;
- for the case $l = \mathsf{cons}(k, l')$ for some integer k and integer list l', we are allowed to **assume** $P(l')$ in the proof of $P(l)$; this assumption is called the **induction hypothesis**.

In summary, we must have **one proof case per formation rule** of the BNF grammar, and reduction is always to the immediate subterms uniquely determined by the formation rule at hand.

The datatype of the **natural numbers** can be represented as terms generated by the BNF grammar:

$$\langle \mathsf{Nat} \rangle ::= \mathsf{zero} \mid \mathsf{succ}(\langle \mathsf{Nat} \rangle)$$

so that, for instance, the natural number 0 is represented by the term zero, and 2 by the term $\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))$. For the natural numbers represented in this way, it is interesting to note that the principle of structural induction coincides with the well-known principle of **mathematical induction** that is usually used for proofs over all natural numbers.

3 Mutually Recursive Datatypes

It is not uncommon that datatypes are defined in terms of each other; such datatypes are called **mutually recursive**. How is the principle of structural induction applied to such datatypes?

Here is an example of mutually recursive datatypes: **integer trees** with arbitrary numbers of children for each node can be represented with a list. We can define this datatype with the following BNF grammar:

$$\begin{array}{rcccl} \langle \mathsf{TreeList} \rangle & ::= & \mathsf{empty} & | & \mathsf{cons}(\langle \mathsf{Tree} \rangle, \langle \mathsf{TreeList} \rangle) \\ \langle \mathsf{Tree} \rangle & ::= & \mathsf{leaf}(\langle \mathsf{Int} \rangle) & | & \mathsf{tree}(\langle \mathsf{TreeList} \rangle) \end{array}$$

We have here two mutually recursive datatypes, TreeList and Tree, each defined by two formation rules that use different constructors (empty and cons for the first datatype, and leaf and tree for the second).

Now, for the datatype TreeList, if we want to define, by structural induction, a function that gives the number of leaves of integer trees, we could introduce a unary function symbol *numleaves*, and write:

$$\begin{array}{rcl} \mathit{numleaves}(\mathsf{empty}) & = & 0 \\ \mathit{numleaves}(\mathsf{cons}(t, tl)) & = & \mathit{numls}(t) + \mathit{numleaves}(tl) \end{array}$$

where we introduce a separate (!) function *numls* for the Tree datatype, which we also define by structural induction:

$$\begin{array}{rcl} \mathit{numls}(\mathsf{leaf}(k)) & = & 1 \\ \mathit{numls}(\mathsf{tree}(tl)) & = & \mathit{numleaves}(tl) \end{array}$$

So, mutually recursive datatypes require mutually recursive functions (and proofs) when using structural induction, one for each datatype. Notice also that the pattern matching in the defining clauses follows (only) the formation rules for the particular datatype over which the function is defined!