Structural Induction in Programming Language Semantics

A Note for C S 386L Students

John A. Thywissen

The University of Texas at Austin

jthywiss@cs.utexas.edu

Programming language syntax and semantics use various structured objects. Languages and derivations (proofs) are two examples. Often, one wishes to prove that a property holds for some set of these structured objects. Typically these objects are defined recursively, so induction is a natural proof approach. Structural induction is one pattern of inductive proof.

In this note, we will (1) define this kind of structured object, then (2) present general induction axiom schemes, then (3) specialize them to induction on languages and derivations, and finally (4) discuss their application in operational semantics.

1. Definitions: Structures, Atoms, Components, and Constituents

We start with some basic definitions of the objects and how they are composed. Following Burstall [2]:

A domain over which structural induction is to be applied is a domain of objects generated by a set of *constructor functions*, which take objects as arguments and yield objects.

An object is either an atom or a structure, further defined as follows.

DEFINITION 1 (Atom). *An atom is an object that was constructed with a nullary constructor function.*

DEFINITION 2 (Structure). *A structure is an object that was constructed with a constructor function given a non-empty, finite sequence of objects as arguments.*

DEFINITION 3 (Component). *An object's components are the arguments given to the the constructor function to create the object.*

We write $A \prec B$ to mean A is a component of B. For example, if object A is constructed via $f(B, C)$, then $B \prec A$ and $C \prec A$. Atoms have no components.

DEFINITION 4 (Constituent). *An object A is a constituent of an object B iff A is identical with B, or if A is a constituent of a component of B.*

This is a relation on objects. We write $A \leq B$ to mean A is a constituent of B. This relation $\leq$ induces a partial order on the objects O. The component relation $\prec$ is the covering relation of the constituent relation $\leq$.[1]

DEFINITION 5 (Proper constituent). *An object A is a proper constituent of an object B iff A is a constituent of B and A is not identical to B,* i.e. $A \leq B \wedge A \neq B$.

We write $A < B$ to mean A is a proper constituent of B.

2. Structural Induction: Strong Form

Burstall presents the structural induction axiom schema as:

AXIOM 1 (Structural Induction, strong). *If, for some set of objects O, an object has a certain property P whenever all its proper constituents have that property, then all the objects in the set have the property.*

$$((\forall a \in O)(\forall b < a)(P(b) \Rightarrow P(a)) \Rightarrow (\forall c \in O)P(c)$$

Note that this is exactly the well-founded induction axiom schema (also known as Noetherian induction or the generalized principle of induction).[2]

Important: Induction is valid iff the partially ordered set $\langle O, < \rangle$ is well-founded: the partially-ordered set

[1] For a refresher on orderings, see the "Order (on a set)" EoM entry [1].

[2] For a introduction to induction see, for example, Genesereth and Kao [3].

must meet the condition that every non-empty, but possibly infinite, subset contains one or more minimal elements. Structures built in the manner described above are well-founded under the proper constituent ordering.

3. Structural Induction: Conventional (Weak) Form

We know that the strong and weak forms of mathematical induction are equivalent. Structural induction has a weak form that is more commonly introduced, as in McCarthy and Painter [5]. This form may seem more similar to conventional mathematical induction, since this form's induction step corresponds to a single application of the constructor function.

AXIOM 2 (Structural Induction, conventional). *If, for some set of objects O, all atoms have a certain property P and all structures have the property P whenever all their components have that property, then all the objects in the set have the property.*

$$((\forall a \in O)(\forall b \prec a)(P(b) \Rightarrow P(a)) \Rightarrow (\forall c \in O)P(c)$$

Note that $(\forall b \prec a)P(b)$ is vacuously true when a is an atom, since it has no components. Simplifying for this case results in the conventional induction base case: For all atoms, just demonstrate $P(a)$.

4. Structural Induction on Languages

Given a grammar, one may wish to demonstrate the validity of some property of terms of the language generated by the grammar.

The structural induction axiom schema can be used quite simply. An object, in this case, is a term of the language. The constructor functions are given by the grammar rules.[3] The proof can proceed by cases, and these cases are usually given directly by the grammar rules.

Some cases will correspond to grammar rules consisting of terminal symbols only, and these are the induction base cases. Other cases will correspond to grammar rules containing one or more nonterminal symbols. In those cases, the inductive hypothesis can be applied to the component nonterminal symbols.

[3] This isn't a direct correspondence for any arbitrary grammar rule. For example, the (somewhat silly) BNF rule $T ::= T$ doesn't correspond to a constructor function. However, abstract syntax grammar in particular is written to have a simple relation to constructor functions. This is assumed, and not further discussed here.

4.1 Example

For example, suppose we take the following language:

⟨*term*⟩ ::= ⟨*term*⟩ + ⟨*term*⟩
| ⟨*term*⟩ ∗ ⟨*term*⟩
| ⟨*literal*⟩

⟨*literal*⟩ ::= `0` | `1`

A proof of a property of this language by induction on the structure of the terms would have four cases:

1. Proof of the property for the term "0". (A base case)
2. Proof of the property for the term "1". (Another base case)
3. Proof of the property for the term "x + y", under the assumption that the property holds for "x" and for "y". (An inductive step)
4. Proof of the property for the term "x * y", under the assumption that the property holds for "x" and for "y". (Another inductive step)

5. Structural Induction on Derivations

Another common type of structure is a derivation of a conclusion using a set of inference rules. Derivations are used to justify, for example, reductions, evaluations, and typing judgments.

5.1 Inference Rules and Derivations

An inference rule system is specified in the form of rules, which are conventionally written:

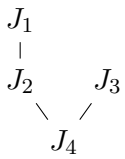
$$\frac{\text{premise} \quad \text{premise} \quad \text{premise...}}{\text{conclusion}}$$

The premises and conclusions are judgements. (Here, judgements are statements such as "t is well-typed" or "t evaluates to v".) If all of the premises of a rule are derivable, then the rule conclusion is derivable.

In inference rule systems such as this, derivations are trees, where the root of the tree is the final conclusion. The interior nodes are judgements used in rules leading to the final conclusion. Leaves are the assumptions or axioms, which appear as rules with no premises. For example, a derivation that uses these rule instances:

$$\frac{}{J_1} \quad \frac{J_1}{J_2} \quad \frac{}{J_3} \quad \frac{J_2 \; J_3}{J_4}$$

can be drawn as the following tree (root at the bottom):



J_4 is the root of the derivation tree. J_1 and J_3 are leaves.

5.2 Induction using Derivations

The structural induction axiom schema can be used quite simply with derivations. An object, in this case, is a derivation tree. The constructor functions are the inference rules, which take the premise derivations as arguments. In other words, an inference rule constructs a derivation tree from zero or more derivation trees.

The proof can proceed by cases, and these cases are given directly by the inference rules. The axioms (rules with no premises) are the induction base cases. The rules with premises can use the induction hypothesis to assume the induction property holds for each of the premises.

5.3 Example

Suppose we are given the following inference system to make yummy judgements over certain breakfast foods:

$$\text{banana yummy} \quad \text{(Banana axiom)}$$

$$\text{Nutella yummy} \quad \text{(Nutella axiom)}$$

$$\frac{x \text{ yummy} \quad y \text{ yummy}}{\text{crêpe}(x, y) \text{ yummy}} \text{ (Crêpe rule)}$$

$$\frac{x \text{ yummy} \quad y \text{ yummy}}{\text{waffle}(x, y) \text{ yummy}} \text{ (Waffle rule)}$$

$$\frac{x \text{ yummy} \quad x \neq \text{Nutella}}{\text{syrup}(x) \text{ yummy}} \text{ (Syrup rule)}$$

Suppose that we wish to prove that if a breakfast food is yummy, then it is edible. A proof by structural induction on derivations would have five cases:

1. Proof of edible for banana yummy. (Banana base case)
2. Proof of edible for Nutella yummy. (Nutella base case)
3. Proof of edible for crêpe(x, y) yummy, under the assumption edible for x yummy and y yummy. (Crêpe inductive step)
4. Proof of edible for waffle(x, y) yummy, under the assumption of edible for x yummy and y yummy. (Waffle inductive step)
5. Proof of edible for the judgement syrup(x) yummy, under the assumption of edible for x yummy and x not being Nutella. (Syrup inductive step)

6. Warning about Case Splits

When a proof uses cases, it is crucial that the cases in aggregate are comprehensive. Omitting a case is an easy mistake to make and would invalidate the entire structural induction. Automated theorem provers and proof assistants are very helpful here because they will point out where the cases don't cover all possible constructions. If proving manually, you must be particularly scrupulous about the correspondence between the cases and the constructors.

7. Operational Semantics

Now that we have described structures and structural induction, let's apply it to programming language semantics.

7.1 Structured Operational Semantics

In Structured Operational Semantics [7] (also called small-step semantics), a set of *configurations* is given, along with a reduces relation ($\longrightarrow$) among the configurations. Conventionally, the reduces relation is specified via inference rules in the form

$$\frac{\gamma_1 \longrightarrow \gamma_1' \quad \gamma_2 \longrightarrow \gamma_2' \quad \cdots}{\gamma \longrightarrow \gamma'}$$

The judgement $\gamma \longrightarrow \gamma'$ asserts that the configurations γ and γ' are in the reduces relation.

The reduces relation is sometimes read as "reduces in one step" to be more explicit. Note that the reduces relation is irreflexive—configurations do not reduce to themselves in one step.

Configurations represent the state of the reduction. This can be a program (language term), or perhaps a program and the state of memory (the store).

Some configurations are designated as *terminal configurations* T. Terminal configurations cannot be further reduced:

$$(\forall \gamma \in T)\ (\nexists \gamma')\ \gamma \longrightarrow \gamma'$$

In the case where configurations are just language terms, the values of the language are the terminal con-

figurations. These are configurations that correspond to a program halting normally.

Not all irreducible configurations are terminal configurations, though. A configuration where no reduction applies, yet it is not a terminal configuration is said to be *stuck*.

The transitive-reflexive closure of reduces, *i.e.* reduces in zero or more steps, is written $\longrightarrow^*$.

Configurations are evaluated by applying series of reduction steps:

$$\gamma \longrightarrow \gamma' \longrightarrow \gamma'' \longrightarrow \gamma''' \longrightarrow \gamma''''$$

this is also

$$\gamma \longrightarrow^* \gamma''''$$

To justify each step, a derivation of that step can be exhibited using the reduction rules of the language. Justification of the reduction sequence $\gamma \longrightarrow \gamma' \longrightarrow \gamma'' \longrightarrow \gamma'''$ would have three derivation trees, with roots $\gamma \longrightarrow \gamma'$, $\gamma' \longrightarrow \gamma''$, and $\gamma'' \longrightarrow \gamma'''$

$$\dfrac{\dfrac{\dots}{\gamma_1 \longrightarrow \gamma_1'} \quad \dfrac{\dots}{\gamma_2 \longrightarrow \gamma_2'}}{\gamma \longrightarrow \gamma'} \quad \dfrac{\dfrac{\dots}{\dots} \quad \dfrac{\dots}{\dots}}{\gamma' \longrightarrow \gamma''} \quad \dots$$

7.2 Natural Semantics

Natural Semantics [4] (also called big-step semantics) specify the evaluates relation, written as $\Rightarrow$ (conventionally) or as $\Downarrow$ (to avoid confusion with implication). This relation is between terms and values $t \Downarrow v$.

Unlike the sequence of steps that Structured Operational Semantics produces in an evaluation, Natural Semantics evaluates *directly* to a value in one step. The evaluation is justified with a single derivation.

7.3 Induction Schemes in Operational Semantics

There are many induction schemes that can be used in various situations when working with operational semantics.

Evaluations in Natural Semantics (big step) are very amenable to structural induction on derivations. Another possibility for Natural Semantics is induction on the depth of the derivation.

However, in Structured Operational Semantics (small step), since an evaluation is a series of steps, structural induction on a derivation only works for a *single* step. An alternative scheme in small step evaluations is induction on the number of steps in an evaluation.

For some properties to be proven, one may be able to use induction on the structure of the language terms, instead of induction on derivations.

It's important to note that structural induction is *not* the only type of induction used in programming language proofs. For example, section 3.3 of the Pierce [6] textbook demonstrates induction on depth of terms and induction on size of terms.

References

[1] Encyclopedia of Mathematics. [Web site]. Available at: http://www.encyclopediaofmath.org/.

[2] BURSTALL, R. M. 1969. Proving properties of programs by structural induction. *The Computer Journal 12,* 1, 41–48.

[3] GENESERETH, M. AND KAO, E. 2012. *Introduction to Logic*. Chapter 9: Induction. Available at: http://logic.stanford.edu/intrologic/chapters/chapter_09.html.

[4] KAHN, G. 1987. Natural semantics. In *Proceedings of 4th Annual Symposium on Theoretical Aspects of Computer Science, STACS 87* (Passau, Germany, 19–21 Feb 1987), F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds. Lecture Notes in Computer Science Series, vol. 247. Springer, 22–39.

[5] MCCARTHY, J. AND PAINTER, J. 1967. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed. Proceedings of Symposia in Applied Mathematics Series, vol. 19. AMS, 33–41.

[6] PIERCE, B. C. 2002. *Types and Programming Languages*. MIT Press.

[7] PLOTKIN, G. D. 2004. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming 60–61*, 17–139. Reprint of: Plotkin, G.D. 1981. "A structural approach to operational semantics". Tech. report DAIMI FN-19. Aarhus University.