輪講：Type Systems and Programming Languages

浜中 信行

2001 年 5 月 7 日

3 Untyped Arithmetic Expressions

3.5 Evaluation

まずは、簡単のため boolean expression の operational semantics についてのみみていく。図 3-1 参照。左側が syntax で、*term* と *value* がそれぞれ定義されている。value とは、それ以上評価されない term である。右側は評価規則で、*evaluation relation* を term 上に定義する。「t → t'」がそれで、「term t を 1 ステップ評価すると t' になる」と読む。
明示的に書かれてはいないが重要なことに、

- true, false はこれ以上評価されない（そのような評価規則はない）。
- If 分は必ず判定部分が先に評価される（E-If による）。

ということがある。
先の 3 つの規則について、E-IfTrue と E-IfFalse は実質的な評価を行い、E-If は評価すべき場所（どの subterm を先に評価するか）を規定している、とみなすことができる。この特徴を強調して、前者・後者それぞれを *computation rule*、*congruence rule* と呼んだりする。

evaluation relation をもう少し formal に定義すると次のようになる。

定義 3.5.1 評価規則の *instance* は、各メタ変数を一貫性を持って term で置き換えることによって得られるものである。このとき 1 ステップの evaluation relation "→" は、図 3-1 の 3 つの評価規則のすべてのインスタンスを含む term 上の"smallest binary relation"となる。ペア (t, t') がこの evaluation relation に含まれる場合、「judgement（statement）t → t' は derivable である」という。
ここで、smallest という言葉により「t → t' が derivable iff 推論規則によって justify される」ということが保証される。この justify は *derivation tree*（導出木）を示すことによってなされる。導出木の例は図 3-3。
「statement t → t' が derivable iff 導出木が存在」という事実は有用で、例えば induction on derivation と呼ばれる technique に直結したりする。例えば次の定理（の証明）などである。

定理 3.5.2 If t → t' and t → t" then t' = t"。
（証明）induction on derivation による。各導出において、最後の導出直前まで望む結果（この場合定理 3.5.2）が成り立っているものと仮定し、最後の導出でもやはり成り立つことを示す。今

考えている例では、図 3-1 の 3 つの評価規則について示せばよい。

ちなみに、定理 3.5.2 の性質を有する言語を deterministic と言う。

次に multi step の evaluation relation をみていく。始めは準備から。

定義 3.5.3 term `t` への評価規則がない（すなわち t → t' となる t' が存在しない）とき、`t` は *normal form* である。

これまでの例では、true、false が normal form となる。この定義について別の言い方をすると次のようになる。

定理 3.5.4 すべての value は normal form である。

言語を拡張するたびに、定理 3.5.4 が成り立つようにしなければならない。そうでない言語は破綻する。

現在考えているシステムでは定理 3.4.5 の逆も成り立つ。しかしながら、これは一般的には成り立つものではない。

定理 3.5.5 term `t` が normal form ならば、`t` は value である。

定義 3.5.6 multi-step evaluation relation $\rightarrow^*$ は、one-step evaluation relation の reflexive, transitive closure である。すなわち次を満たす smallest relation である。

- if t → t' then t $\rightarrow^*$ t'
- t $\rightarrow^*$ t for all t
- if t $\rightarrow^*$ t' and t' $\rightarrow^*$ t" then t $\rightarrow^*$ t"

定理 3.5.8 If t $\rightarrow^*$ u and t $\rightarrow^*$ u', where u and u' are both normal forms, then u = u'.

要するに、どのように評価しようとその結果は一様であることを示している（結果（normal form）があるならば）。証明は induction on derivation を利用して行なう（cf. 定理 3.5.2）。

次の性質も、必ずしもすべての言語で成り立つ必要のないものである。

定理 3.5.9 For every term `t` there is some value `v` such that `t` $\rightarrow^*$ v.

（証明）定理 3.5.8 より、term は normal form まで評価できるならばその結果は一様である。また、定理 3.5.5 より normal from は value である。次に、すべての term が normal form まで評価できるかであるが、各 evaluation step は term の size を小さくしていることより保証される。

次に、これまでのものを拡張して arithmetic expression の評価を加える。図 3-2 のようになる（拡張分）。図 3-1 と比べて、*numeric value* と呼ばれる区分があることであり、評価規則にも制限を加えている（図 3-4）。

言語の operational semantics を形式化しようとする場合、"succ false" といった正しくない場合の挙動も明確にする必要がある。

定義 3.5.12 term は normal form であるが value でない場合、*stuck* しているという。

”stuckness”は、runtime error の概念を与えてくれる。直観的には、プログラムが意味のない状態に入ってしまい、operational semantics が何をすべきか規定していない状態、とまとめられる。

意味のない状態を形式的に扱う別の方法として、*wrong* という term を用意し、stuck するような場合はすべて wrong に評価されるような評価規則を加える、というようなものもある（図 3-6、exercise3.5.13）。

最後に、operational semantics には、*small step* と *big step* という 2 つのスタイルがある。small step semantics はこれまで見てきたような、一つ一つの term の書き換えを規定するようなものである。対して big step semantics は、「この term は最終的にこの value になる」といったことを直接形式化したものである。big step semantics の評価規則は図 3-5 のようになる。もちろん、small step semantics は一致する（☆が成立、exercise3.5.14）。

4 An ML implmenetation of Arithmetic Expressions

この章の意図はよくわからないのだが、書いてあることから推察すると「ML で先ほど定義した Boolean、Arithmetic Exspressions の評価系を記述しよう」とでもなるのか。とりあえず、ここでは実装に用いる言語への用件として、

- 自動メモリ管理機能を備えている
- データ構造に関するマッチングを使った再帰関数の定義が簡単に行なえる

という 2 点をあげている。その上でここでの実装言語を Objective Caml としている（他では Standard ML、Haskell、Scheme は良い選択で、Java や C はいまいちと述べられている）。

4.1 Syntax

term の定義は図 4-1 の通り。datatype の定義に「info」というものが見られるが、parser が情報埋め込む情報で、デバッグなどに使うことができる。省略可能である。

図 4-1 の定義に対し、term が numeric number かどうか調べる関数、value かどうか調べる関数は、それぞれ図 4-2、4-3 となる。

4.2 Evaluation

先で定義した term を、図 3-1、図 3-2 の評価規則にのっとって評価する関数は図 4-4 のようになる（`eval1`）。図 3-1、3-2 には、value に対する評価規則がなかった（結果がない）が、実装においては何らかの結果を用意する必要があり、ここでは Exception を投げることにしている。

注意すべき点として、TmPred の評価のところで `when` 節が現れている。これは、例えば nv1 が数値でなければならないのだが ML のマッチングではどの term にもマッチしてしまい、これを防ぐためである。この制限は図 3-1、図 3-2 の評価規則はにないが、図 4-5 のように書き換えてもよい。

図 4-4 の`eval1`を用いて、term を 1 ステップづつ評価する関数は図 4-6 の`eval`となる。eval1 が term を返すたびにそれをまた eval1 で評価し、Exception が投げられたところでその term を返している。

4.3 The Rest of the Stroy

実際のインタプリタやコンパイラには、上で述べたものの他に多くの parts が必要である。ファイルからの入力文字列の解析、解析結果の parsing、評価結果の出力などである。興味のある読者は、OCaml のソースを見てみるとよい、と述べられている。