An Example of Simultaneous Inductive Definitions: Parsing Arithmetical Expressions

Ralph Matthes
Institut für Informatik der Universität München
Oettingenstraße 67, 80538 München
matthes@informatik.uni-muenchen.de

November 12, 2002

Abstract

Section 2 of Lambda Calculus: A Case for Inductive Definitions, course material for the 12th European Summer School in Logic, Language and Information (ESSLLI 2000) in Birmingham (6-18 August 2000) is reprinted.

In theoretical computer science, inductive definitions are ubiquitous. Mostly, they appear in the disguise of formal grammars which are idealizations of natural language grammars. Therefore, this introduction to inductive definitions concentrates on examples of (even context-free) grammars. In [ASU86, Example 4.8] two equivalent grammars for arithmetic expressions are studied for the purpose of illustrating the elimination of left recursion. The first one is (with **id** a set of identifiers):

$$\begin{array}{lcl} \mathsf{E} & \to & \mathsf{E} + \mathsf{T} \mid \mathsf{T} \\ \mathsf{T} & \to & \mathsf{T} * \mathsf{F} \mid \mathsf{F} \\ \mathsf{F} & \to & (\mathsf{E}) \mid \mathbf{id} \end{array}$$

The intuition of the non-terminals is given by:

- E: expression = a sum of terms
- T: term = a product of factors
- F: factor = a parenthesized expression or an identifier

Thus we model that in arithmetic expressions, $*$ has a higher precedence than $+$, and that parentheses enforce grouping. The grammar is left recursive and therefore cannot be treated by top-down parsing methods: One runs into the loop $\mathsf{E} \to \mathsf{E} + \mathsf{T} \to \mathsf{E} + \mathsf{T} + \mathsf{T} \to \mathsf{E} + \mathsf{T} + \mathsf{T} + \mathsf{T} \to \dots$

The second grammar is as follows (with empty word ϵ):

$$\begin{array}{lcl} \mathsf{E} & \to & \mathsf{TH} \\ \mathsf{H} & \to & +\mathsf{TH} \mid \epsilon \\ \mathsf{T} & \to & \mathsf{FK} \\ \mathsf{K} & \to & *\mathsf{FK} \mid \epsilon \\ \mathsf{F} & \to & (\mathsf{E}) \mid \mathbf{id} \end{array}$$

The idea is to decompose an expression into a term and a string of the form $+\mathsf{T}+\mathsf{T}+\ldots+\mathsf{T}$ (expressed by the auxiliary non-terminal H) while in the first grammar, the decomposition would be into a list of T's with $+$'s inbetween. The same idea is used for the terms.

The claim is that both grammars are equivalent, i. e., that the same expressions can be derived from the non-terminal E. How can we prove that with full mathematical rigour? Answer: By induction. More concretely: By induction on the generation of the expressions according to these grammars. This generation process may be made precise as follows: We simultaneously define the sets $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$ of strings derivable in the first grammar from the non-terminals E, T and F, respectively. This will be done by means of an inductive definition to be expressed by the following rules:

(1) If $e \in \mathcal{E}$ and $t \in \mathcal{T}$ then $e + t \in \mathcal{E}$.

(2) If $t \in \mathcal{T}$ then $t \in \mathcal{E}$.

(3) If $t \in \mathcal{T}$ and $f \in \mathcal{F}$ then $t * f \in \mathcal{T}$.

(4) If $f \in \mathcal{F}$ then $f \in \mathcal{T}$.

(5) If $e \in \mathcal{E}$ then $(e) \in \mathcal{F}$.

(6) If $i \in \mathbf{id}$ then $i \in \mathcal{F}$.

These rules have to be interpreted as describing the generation of the sets $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$, hence the three sets will have all the properties expressed by the rules, and an element only enters one of the three sets if this is possible by one of the six rules. E. g., $e + t$ only enters $\mathcal{E}$ by the first rule, if already $e \in \mathcal{E}$ and $t \in \mathcal{T}$.

Clearly, we are only interested in the set $\mathcal{E}$ of expressions, but we have to define sets for every non-terminal in the grammar.

Analogously, we inductively define five sets $\mathcal{E}', \mathcal{H}', \mathcal{T}', \mathcal{K}', \mathcal{F}'$, corresponding to the five non-terminals of the second grammar:

(a) If $t \in \mathcal{T}'$ and $h \in \mathcal{H}'$ then $th \in \mathcal{E}'$.

(b) If $t \in \mathcal{T}'$ and $h \in \mathcal{H}'$ then $+th \in \mathcal{H}'$.

(c) $\epsilon \in \mathcal{H}'$.

(d) If $f \in \mathcal{F}'$ and $k \in \mathcal{K}'$ then $fk \in \mathcal{T}'$.

(e) If $f \in \mathcal{F}'$ and $k \in \mathcal{K}'$ then $*fk \in \mathcal{K}'$.

(f) $\epsilon \in \mathcal{K}'$.

(g) If $e \in \mathcal{E}'$ then $(e) \in \mathcal{F}'$.

(h) If $i \in \mathbf{id}$ then $i \in \mathcal{F}'$.

Lemma 1 (Equivalence) $\mathcal{E} = \mathcal{E}'$.

Proof It is obvious that we cannot prove $\mathcal{E} = \mathcal{E}'$ in isolation. We first prove $\mathcal{E} \subseteq \mathcal{E}'$ and simultaneously prove $\mathcal{T} \subseteq \mathcal{T}'$ and $\mathcal{F} \subseteq \mathcal{F}'$ by induction on the simultaneous inductive definition of $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$. Proving by induction means arguing on the generation of all the strings in $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$ by help of the rules (1) to (6). Since the rules specify which strings have to be in $\mathcal{E}$, $\mathcal{T}$ or $\mathcal{F}$ before, we may assume that for those strings we already have that they are in $\mathcal{E}'$, $\mathcal{T}'$ and $\mathcal{F}'$, respectively. This is always called the induction hypothesis.

We go through (1) to (6):

(1) If $e + t \in \mathcal{E}$ has been concluded from $e \in \mathcal{E}$ and $t \in \mathcal{T}$, we have to show that $e + t \in \mathcal{E}'$. By induction hypothesis, $e \in \mathcal{E}'$ and $t \in \mathcal{T}'$. Since there is only one rule for $\mathcal{E}'$, we conclude that $e = t'h$ with $t' \in \mathcal{T}'$ and $h \in \mathcal{H}'$. Now, we first prove an auxiliary statement: If $h \in \mathcal{H}'$ and $t \in \mathcal{T}'$ then $h + t \in \mathcal{H}'$. This is proved by induction on $\mathcal{H}'$: If $h \in \mathcal{H}'$ due to rule (b) then $h = +t'h'$ with $t' \in \mathcal{T}'$ and $h' \in \mathcal{H}'$ which entered $\mathcal{H}'$ before h. By induction hypothesis, $h' + t \in \mathcal{H}'$, hence $h + t = +t'h' + t \in \mathcal{H}'$ by rule (b) again. If $h \in \mathcal{H}'$ due to rule (c) then $h = \epsilon$. By rules (c) and (b), $h + t = \epsilon + t = +t = +t\epsilon \in \mathcal{H}'$. By help of this statement, we infer that $h + t \in \mathcal{H}'$, hence $e + t = t'h + t \in \mathcal{E}'$ by rule (a).

(2) If $t \in \mathcal{E}$ has been concluded from $t \in \mathcal{T}$, we have to show that $t \in \mathcal{E}'$, and the induction hypothesis is $t \in \mathcal{T}'$. We are done by rules (c) and (a).

(3) We need another auxiliary lemma: If $k \in \mathcal{K}'$ and $f \in \mathcal{F}'$ then $k * f \in \mathcal{K}'$. The proof and the whole case are similar to (1).

(4) Similar to (2).

(5) If $(e) \in \mathcal{F}$ has been concluded from $e \in \mathcal{E}$, we have to show that $(e) \in \mathcal{F}'$, and the induction hypothesis is $e \in \mathcal{E}'$. So were are done by rule (g).

(6) If $i \in \mathcal{F}$ has been concluded from $i \in \mathbf{id}$, we have to show that $i \in \mathcal{F}'$. This holds by rule (h).

Let us prove $\mathcal{E}' \subseteq \mathcal{E}$. Since the definitions of $\mathcal{E}'$, $\mathcal{H}'$, $\mathcal{T}'$, $\mathcal{K}'$ and $\mathcal{F}'$ are entangled, we need to prove something for all of those. Therefore, we inductively define two auxiliary sets $\mathcal{H}$ and $\mathcal{K}$. $\mathcal{H}$ is defined by mimicking rules (b) and (c):

(b+) If $t \in \mathcal{T}$ and $h \in \mathcal{H}$ then $+th \in \mathcal{H}$.

(c+) $\epsilon \in \mathcal{H}$.

Analogously, $\mathcal{K}$ is defined after the model of (e) and (f):

(e+) If $f \in \mathcal{F}$ and $k \in \mathcal{K}$ then $*fk \in \mathcal{K}$.

(f+) $\epsilon \in \mathcal{K}$.

Now, we can simultaneously prove $\mathcal{E}' \subseteq \mathcal{E}$, $\mathcal{H}' \subseteq \mathcal{H}$, $\mathcal{T}' \subseteq \mathcal{T}$, $\mathcal{K}' \subseteq \mathcal{K}$ and $\mathcal{F}' \subseteq \mathcal{F}$ by induction on the simultaneous inductive definition of $\mathcal{E}'$, $\mathcal{H}'$, $\mathcal{T}'$, $\mathcal{K}'$ and $\mathcal{F}'$. In order to do this, we first prove that (a) to (h) hold when all the primed entities $\mathcal{X}'$ are replaced by the $\mathcal{X}$, for $\mathcal{X} \in \{\mathcal{E}, \mathcal{H}, \mathcal{T}, \mathcal{K}, \mathcal{F}\}$.

(a) Show that if $t \in \mathcal{T}$ and $h \in \mathcal{H}$ then $th \in \mathcal{E}$. We need an auxiliary lemma: If $e \in \mathcal{E}$ and $e' \in \mathcal{E}$ then $e + e' \in \mathcal{E}$. It is proved by induction on the generation of $e' \in \mathcal{E}$: If $e' \in \mathcal{E}$ has been concluded by rule (1) then $e' = e'' + t$ for some previously found $e'' \in \mathcal{E}$ and some $t \in \mathcal{T}$. By induction hypothesis, $e + e'' \in \mathcal{E}$, hence by rule (1), $e + e' = e + e'' + t \in \mathcal{E}$. If $e' \in \mathcal{E}$ has been concluded by rule (2) from $e' \in \mathcal{T}$ then $e + e' \in \mathcal{E}$ by (1).

    Now, the claim is proved for arbitrary $t \in \mathcal{T}$ by induction on $h \in \mathcal{H}$: If $h \in \mathcal{H}$ stems from rule (b+) then $h = +t'h'$ for some $t \in \mathcal{T}$ and some previously generated $h' \in \mathcal{H}$. By induction hypothesis, $t'h' \in \mathcal{E}$. Since $t \in \mathcal{E}$, the auxiliary lemma yields $th = t + t'h' \in \mathcal{E}$. If $h \in \mathcal{H}$ by rule (c+) then $h = \epsilon$ and therefore $th = t \in \mathcal{E}$.

(b) By rule (b+): If $t \in \mathcal{T}$ and $h \in \mathcal{H}$ then $+th \in \mathcal{H}$.

(c) $\epsilon \in \mathcal{H}$ by rule (c+).

(d) Show that if $f \in \mathcal{F}$ and $k \in \mathcal{K}$ then $fk \in \mathcal{T}$. In analogy to (a), we first prove $t, t' \in \mathcal{T} \Rightarrow t * t' \in \mathcal{T}$ by induction on $t' \in \mathcal{T}$ and then do induction on $k \in \mathcal{K}$.

(e) By rule (e+): If $f \in \mathcal{F}$ and $k \in \mathcal{K}$ then $*fk \in \mathcal{K}$.

(f) $\epsilon \in \mathcal{K}$ by rule (f+).

(g) By rule (5): If $e \in \mathcal{E}$ then $(e) \in \mathcal{F}$.

(h) If $i \in \mathbf{id}$ then $i \in \mathcal{F}$ by (6).

Therefore, the unprimed entities fulfill the defining clauses of the primed entities. Since the primed entities are assumed to be the smallest sets with those properties (elements only enter if one of the rules applies), we conclude that the primed entities are all included in the unprimed ones, respectively.

Readers not being familiar with this order-theoretic arguments may simply do an inductive proof of $x \in \mathcal{X}' \Rightarrow x \in \mathcal{X}$ for $\mathcal{X} \in \{\mathcal{E}, \mathcal{H}, \mathcal{T}, \mathcal{K}, \mathcal{F}\}$ in traditional style. We only consider this for rule (a): Assume that $th \in \mathcal{E}'$ has been concluded from $t \in \mathcal{T}'$ and $h \in \mathcal{H}'$. By induction hypothesis, $t \in \mathcal{T}$ and $h \in \mathcal{H}$. Therefore, by the validity of (a) with $\mathcal{T}'$, $\mathcal{H}'$ and $\mathcal{E}'$ replaced by $\mathcal{T}$, $\mathcal{H}$ and $\mathcal{E}$, respectively, we get $th \in \mathcal{E}$ and are done. □

Since our proof only contained constructive arguments, it is straightforward to write a program by which the parse trees for expressions according to the first and the second grammar can be transformed into each other. The programming language we chose is Standard ML of New Jersey (Version 110.0.6 of October 1999).

The readers are not expected to be familiar with SML. Nevertheless, it should be possible to understand the program text without explaining the syntax. Only the intentions are given and the output of the system when fed with the program lines shown so far.

```
datatype
    'id expr = Pnode (* P for plus *) of 'id expr * 'id term |
               Tnode (* T for term *) of 'id term
and
    'id term = Mnode (* M for multiply *) of 'id term * 'id factor |
               Fnode (* F for factor *) of 'id factor
and
    'id factor = Bnode (* B for bracket *) of 'id expr |
                 Inode (* I for identifier *) of 'id;
```

Three inductive datatypes are introduced which are parameterized by `'id` representing the set of identifiers (in our examples this will always be the built-in type `string` of strings). They model the set of derivation trees for $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$. The rules (1) to (6) are represented by constructors (which are nothing but tags). So, e. g., Pnode stands for (1) and thus requires some `'id expr` and some `'id term` (`*` denotes pairing). Note that only the derivation trees are modeled and not the expressions themselves. The system's answer (with its preferred type variable name `'a`):

```
datatype 'a expr = Pnode of 'a expr * 'a term | Tnode of 'a term
datatype 'a term = Fnode of 'a factor | Mnode of 'a term * 'a factor
datatype 'a factor = Bnode of 'a expr | Inode of 'a
```

```
val f1 = Bnode(Pnode(Tnode(Fnode(Inode("7"))),Fnode(Inode("3"))));

val t1 = Fnode(Bnode(Tnode(Mnode(Fnode(Inode("8")),f1))));

val e1 = Pnode(Tnode(t1),Mnode(Fnode(Inode("6")),Inode("5")));

fun itplus(expr,n) =
    if n <= 0 then expr
    else Pnode(expr,Fnode(Bnode(itplus(expr,n-1))));
    (* for the formation of big terms *)

val e2 = itplus(e1,50);
val e3 = itplus(e1,1000);
val e4 = itplus(e1,100000);
```

Some examples are provided, leading to the output

```
val f1 = Bnode (Pnode (Tnode (Fnode #),Fnode (Inode #))) : string factor
val t1 = Fnode (Bnode (Tnode (Mnode (#,#)))) : string term
val e1 = Pnode (Tnode (Fnode (Bnode #)),Mnode (Fnode #,Inode #)) : string expr
val itplus = fn : 'a expr * int -> 'a expr
val e2 = Pnode (Pnode (Tnode #,Mnode #),Fnode (Bnode (Pnode #))) : string expr
val e3 = Pnode (Pnode (Tnode #,Mnode #),Fnode (Bnode (Pnode #))) : string expr
val e4 = Pnode (Pnode (Tnode #,Mnode #),Fnode (Bnode (Pnode #))) : string expr
```

Note that # indicates truncation of output. By recursion on the derivation trees we define the derived elements of $\mathcal{E}$, $\mathcal{T}$ and $\mathcal{F}$ as lists of symbols:

```
fun ppe(Pnode(expr,term)) = ppe(expr)@["+"]@ppt(term) |
    ppe(Tnode(term)) = ppt(term)

and
    ppt(Mnode(term,factor)) = ppt(term)@["*"]@ppf(factor) |
    ppt(Fnode(factor)) = ppf(factor)

and
    ppf(Bnode(expr)) = ["("]@ppe(expr)@[")"] |
    ppf(Inode(ident)) = [ident];

val str1 = ppf f1;
val str2 = ppt t1;
val str3 = ppe e1;
```

Note that `@` denotes concatenation of lists. We get

```
val ppe = fn : string expr -> string list
val ppt = fn : string term -> string list
val ppf = fn : string factor -> string list
val str1 = ["(","7","+","3",")"] : string list
val str2 = ["(","8","*","(","7","+","3",")",")"] : string list
val str3 = ["(","8","*","(","7","+","3",")",")","+","6","*",...] : string list
```

Note that `...` denotes truncation of the list (although in this example `...` stands for nothing but `"5"` which can be checked by entering the line `(str3=["(","8","*","(","7","+","3",")",")","+","6","*","5"]);` the response to which should be `val it = true : bool`).

The second grammar is dealt with similarly (see next page), yielding

```
datatype 'a ee = THnode of 'a tt * 'a hh
datatype 'a hh = EmptyH | PPnode of 'a tt * 'a hh
datatype 'a tt = FKnode of 'a ff * 'a kk
datatype 'a kk = EmptyK | MMnode of 'a ff * 'a kk
datatype 'a ff = BBnode of 'a ee | IInode of 'a
val hh1 = PPnode (FKnode (IInode #,MMnode #),EmptyH) : string hh
val kk1 = MMnode (BBnode (THnode (#,#)),EmptyK) : string kk
val ee1 = THnode (FKnode (BBnode #,EmptyK),PPnode (FKnode #,EmptyH))
  : string ee
val ppee = fn : string ee -> string list
val pphh = fn : string hh -> string list
val pptt = fn : string tt -> string list
val ppkk = fn : string kk -> string list
val ppff = fn : string ff -> string list
```

```
datatype
    'id ee = THnode of 'id tt * 'id hh
and
    'id hh = PPnode of 'id tt * 'id hh |
             EmptyH
and
    'id tt = FKnode of 'id ff * 'id kk
and
    'id kk = MMnode of 'id ff * 'id kk |
             EmptyK
and
    'id ff = BBnode of 'id ee |
             IInode of 'id;

(* examples *)
val hh1 = PPnode(FKnode(IInode("6"),MMnode(IInode("5"),EmptyK)),EmptyH);

val kk1 = MMnode(BBnode(THnode(FKnode(IInode("7"),EmptyK),
                PPnode(FKnode(IInode("3"),EmptyK),EmptyH))),EmptyK);

val ee1 = THnode(FKnode(BBnode(THnode(FKnode(IInode("8"),kk1),
                                      EmptyH)),EmptyK),hh1);

(* string representation *)
fun ppee(THnode(tt,hh)) = pptt(tt)@pphh(hh)
and
    pphh(PPnode(tt,hh)) = ["+"]@pptt(tt)@pphh(hh) |
    pphh(EmptyH) = nil
and
    pptt(FKnode(ff,kk)) = ppff(ff)@ppkk(kk)
and
    ppkk(MMnode(ff,kk)) = ["*"]@ppff(ff)@ppkk(kk) |
    ppkk(EmptyK) = nil
and
    ppff(BBnode(ee)) = ["("]@ppee(ee)@[")"] |
    ppff(IInode(ident)) = [ident];
```

```
val str4 = ppee(ee1);
val ok1 = (str3 = str4);
```

leads to

```
val str4 = ["(","8","*","(","7","+","3",")",")","+","6","*",...] : string list
val ok1 = true : bool
```

and shows that `ee1` is the derivation in the second grammar of the same expression as that derived by `e1` in the first one.

Now we come to the conversion from the first grammar to the second grammar reflecting the first part of the proof of the lemma. The auxiliary statement for (1), namely if $\mathrm{h} \in \mathcal{H}'$ and $\mathrm{t} \in \mathcal{T}'$ then $\mathrm{h} + \mathrm{t} \in \mathcal{H}'$, is turned into a function `plus` which takes an argument in `'a hh` and one in `'a tt` to produce that in `'a hh` reflecting their addition. Similarly, we define `mult` to reflect the auxiliary lemma for (3). Moreover, we exploit that there are only one rule for $\mathcal{E}'$ and $\mathcal{T}'$ each which allows to decompose the derivations into two pieces.

```
fun plus(PPnode(tt',hh),tt) = PPnode(tt',plus(hh,tt)) |
    plus(EmptyH,tt) = PPnode(tt,EmptyH);

fun mult(MMnode(ff',kk),ff) = MMnode(ff',mult(kk,ff)) |
    mult(EmptyK,ff) = MMnode(ff,EmptyK);

fun eetott(THnode(tt,hh)) = tt;

fun eetohh(THnode(tt,hh)) = hh;

fun tttoff(FKnode(ff,kk)) = ff;

fun tttokk(FKnode(ff,kk)) = kk;
```

```
val plus = fn : 'a hh * 'a tt -> 'a hh
val mult = fn : 'a kk * 'a ff -> 'a kk
val eetott = fn : 'a ee -> 'a tt
val eetohh = fn : 'a ee -> 'a hh
val tttoff = fn : 'a tt -> 'a ff
val tttokk = fn : 'a tt -> 'a kk
```

The following definitions of the transforming functions are straightforwardly read off the proofs of (1) to (6). Clearly, they have to be defined simultaneously, and every application of the induction hypothesis now becomes some recursive call to one of the three functions about to be defined.

```
fun cve(Pnode(expr,term)) =
    let val ee = cve(expr) in
        THnode(eetott(ee),plus(eetohh(ee),cvt(term)))
    end |
    cve(Tnode(term)) = THnode(cvt(term),EmptyH)

and
    cvt(Mnode(term,factor)) =
    let val tt = cvt(term) in
        FKnode(tttoff(tt),mult(tttokk(tt),cvf(factor)))
    end |
    cvt(Fnode(factor)) = FKnode(cvf(factor),EmptyK)

and
    cvf(Bnode(expr)) = BBnode(cve(expr))|
    cvf(Inode(ident)) = IInode(ident);

val ok2=(cve(e1)=ee1);
```

Notice that the `let` construction provides for abbreviations used locally.

```
val cve = fn : 'a expr -> 'a ee
val cvt = fn : 'a term -> 'a tt
val cvf = fn : 'a factor -> 'a ff
val ok2 = true : bool
```

We turn to the other direction corresponding to the proof of $\mathcal{E}' \subseteq \mathcal{E}$. The two auxiliary sets $\mathcal{H}$ and $\mathcal{K}$ are represented by two additional inductive datatypes:

```
datatype    'id auxh = PPPnode of 'id term * 'id auxh |
    EmptyAuxH;

datatype    'id auxk = MMMnode of 'id factor * 'id auxk |
    EmptyAuxK;
```

```
datatype 'a auxh = EmptyAuxH | PPPnode of 'a term * 'a auxh
datatype 'a auxk = EmptyAuxK | MMMnode of 'a factor * 'a auxk
```

We again need auxiliary functions reflecting the lemmas

- If $e \in \mathcal{E}$ and $e' \in \mathcal{E}$ then $e + e' \in \mathcal{E}$.
- If $t \in \mathcal{T}$ and $h \in \mathcal{H}$ then $th \in \mathcal{E}$.
- If $t \in \mathcal{T}$ and $t' \in \mathcal{T}$ then $t * t' \in \mathcal{T}$.
- If $f \in \mathcal{F}$ and $k \in \mathcal{K}$ then $fk \in \mathcal{T}$.

embedded into the proof. The operations on the derivations are given in the order of those lemmas:

```
fun pluse(expr,Pnode(expr',term)) = Pnode(pluse(expr,expr'),term) |
    pluse(expr,Tnode(term)) = Pnode(expr,term);

fun concth(term,EmptyAuxH) = Tnode(term) |
    concth(term,PPPnode(term',auxh)) =
          pluse(Tnode(term),concth(term',auxh));


fun multt(term,Mnode(term',factor)) = Mnode(multt(term,term'),factor) |
    multt(term,Fnode(factor)) = Mnode(term,factor);

fun concfk(factor,EmptyAuxK) = Fnode(factor) |
    concfk(factor,MMMnode(factor',auxk)) =
          multt(Fnode(factor),concfk(factor',auxk));
```

```
val pluse = fn : 'a expr * 'a expr -> 'a expr
val concth = fn : 'a term * 'a auxh -> 'a expr
val multt = fn : 'a term * 'a term -> 'a term
val concfk = fn : 'a factor * 'a auxk -> 'a term
```

As before, no intuition is needed to produce the transforming functions from the proofs of (a) to (h). However, note that since we only showed them for the rules where the primed entities $\mathcal{X}'$ have been replaced by the $\mathcal{X}$, the applications of the induction hypotheses are hidden, but nevertheless the recursive calls have to be made as for the other direction.

```
fun cvee(THnode(tt,hh)) = concth(cvtt(tt),cvhh(hh))

and
    cvhh(PPnode(tt,hh)) = PPPnode(cvtt(tt),cvhh(hh)) |
    cvhh(EmptyH) = EmptyAuxH
and
    cvtt(FKnode(ff,kk)) = concfk(cvff(ff),cvkk(kk))
and
    cvkk(MMnode(ff,kk)) = MMMnode(cvff(ff),cvkk(kk)) |
    cvkk(EmptyK) = EmptyAuxK
and
    cvff(BBnode(ee)) = Bnode(cvee(ee)) |
    cvff(IInode(ident)) = Inode(ident);

val ok3 = (cvee(ee1) = e1);

val ok4 = (cvee(cve(e2)) = e2);

val ok5 = (cvee(cve(e3)) = e3);

val ok6 = (cvee(cve(e4)) = e4); (* takes some time *)
```

```
val cvee = fn : 'a ee -> 'a expr
val cvhh = fn : 'a hh -> 'a auxh
val cvtt = fn : 'a tt -> 'a term
val cvkk = fn : 'a kk -> 'a auxk
val cvff = fn : 'a ff -> 'a factor
val ok3 = true : bool
val ok4 = true : bool
val ok5 = true : bool
val ok6 = true : bool
val it = () : unit
```

Note that the last line of output stems from the execution of the whole program loaded into SML by the command `use("expr.sml")` where `expr.sml` is the name of the source file.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, 1986.