9.1 Numeric Terms

9.1.1 Introduction. The symbolic data structures are usually defined in the functional programming languages with the help of union types which can be readily arithmetized. We have seen an example of union type defining binary trees in Sect. 8.1. We will use in the following paragraphs another union type to arithmetize a certain class of expressions.

Suppose that we wish to operate symbolically on numeric terms which are formed from variables x_i, constants n by the numeric operators + (addition) and × (multiplication). Functional programming languages use the following union type to specify the domain of numeric terms:

$$\mathit{Term} = \mathit{Var}(\mathrm{N}) \mid \mathit{Const}(\mathrm{N}) \mid \mathit{Add}(\mathit{Term}, \mathit{Term}) \mid \mathit{Mult}(\mathit{Term}, \mathit{Term}).$$

A value of type *Term* is therefore either a variable $\mathit{Var}(i)$, or a constant $\mathit{Const}(n)$, or an addition $\mathit{Add}(t_1, t_2)$, or a multiplication $\mathit{Mult}(t_1, t_2)$, where i and n are of type N and t_1 and t_2 are values of type *Term*. The functions $\mathit{Var}(i)$, $\mathit{Const}(n)$, $\mathit{Add}(t_1, t_2)$ and $\mathit{Mult}(t_1, t_2)$ are called *constructors*.

9.1.2 Constructors of numeric terms. Arithmetization of numeric expressions is done with the help of the following four pair constructors with pairwise different tags:

$$\begin{aligned} \mathrm{x}_i^\bullet &= \langle 0, i \rangle && \text{(variables)} \\ n^\bullet &= \langle 1, n \rangle && \text{(constants)} \\ t_1 +^\bullet t_2 &= \langle 2, t_1, t_2 \rangle && \text{(addition)} \\ t_1 \times^\bullet t_2 &= \langle 3, t_1, t_2 \rangle. && \text{(multiplication)} \end{aligned}$$

From the properties of the pairing function we obtain the constructors are pairwise disjoint and that the constructors are injective mappings, e.g.

$$\begin{aligned} &\vdash_{\mathrm{PA}} \mathrm{x}_i^\bullet \neq t_1 +^\bullet t_2 \\ &\vdash_{\mathrm{PA}} t_1 \times^\bullet t_2 = t_1' \times^\bullet t_2' \to t_1 = t_1' \land t_2 = t_2'. \end{aligned}$$

Similar properties hold also for the other constructors.

The pattern matching style of definitions of functions operating over the codes of numeric terms is obtained with the conditionals of the form

$$\begin{array}{l} \textbf{case} \\ \quad t = \mathrm{x}_i^\bullet \Rightarrow_i \beta_1[i] \\ \quad t = n^\bullet \Rightarrow_n \beta_2[n] \\ \quad t = t_1 +^\bullet t_2 \Rightarrow_{t_1, t_2} \beta_3[t_1, t_2] \\ \quad t = t_1 \times^\bullet t_2 \Rightarrow_{t_1, t_2} \beta_4[t_1, t_2] \\ \quad \textbf{otherwise} \Rightarrow \beta_5. \\ \textbf{end} \end{array}$$

This is called *discrimination on the constructors of numeric terms.*

The above conditional is evaluated as follows. Consider, for instance, its third variant $t = t_1 +^\bullet t_2$. The expression

$$Tuple_*(3,t) \land_* [t]^3_1 =_* 2$$

is its characteristic term as we have

$$\vdash_{PA} \exists t_1 \exists t_2\, t = t_1 +^\bullet t_2 \leftrightarrow Tuple(3,t) \land [t]^3_1 = 2.$$

Note also that

$$\vdash_{PA} t = t_1 +^\bullet t_2 \to t_1 = [t]^3_2 \land t_2 = [t]^3_3$$

and therefore, the terms $[t]^3_2$ and $[t]^3_3$ are the witnessing terms for the output variables t_1, t_2 of this variant. Similarly for the other variants.

9.1.3 Arithmetization of numeric terms. We wish to assign to every numeric τ a unique number $\ulcorner\tau\urcorner$, called *the code of* τ. The mapping is defined inductively on the structure of numeric terms:

$$\begin{aligned}
\ulcorner x_i \urcorner &= \mathrm{x}^\bullet_i \\
\ulcorner n \urcorner &= n^\bullet \\
\ulcorner \tau_1 + \tau_2 \urcorner &= \ulcorner \tau_1 \urcorner +^\bullet \ulcorner \tau_1 \urcorner \\
\ulcorner \tau_1 \times \tau_2 \urcorner &= \ulcorner \tau_1 \urcorner \times^\bullet \ulcorner \tau_1 \urcorner.
\end{aligned}$$

We can now encode, for instance, the term $4 \times x_5 + x_7$ by the number

$$4^\bullet \times^\bullet \mathrm{x}^\bullet_5 +^\bullet \mathrm{x}^\bullet_7 = 103\,635\,707\,473\,048\,605\,704.$$

Discrimination on the constructors of numeric terms is used in the definition of the p.r. predicate $Term(t)$ holding of the codes of numeric terms. The predicate is defined by course of values recursion as follows:

$$\begin{aligned}
&Term(\mathrm{x}^\bullet_i) \\
&Term(n^\bullet) \\
&Term(t_1 +^\bullet t_2) \leftarrow Term(t_1) \land Term(t_2) \\
&Term(t_1 \times^\bullet t_2) \leftarrow Term(t_1) \land Term(t_2).
\end{aligned}$$

In the sequel we identify numeric terms with their codes and from now on we will say *the numeric term* t instead of *the code* t *of a numeric term.*

9.1.4 Case analysis on numeric terms. From the definition of the predicate *Term* we get directly the following property

$$\vdash_{PA} Term(t) \to \exists i\, t = \mathrm{x}^\bullet_i \lor \exists n\, t = n^\bullet \lor \exists t_1 \exists t_2\, t = t_1 +^\bullet t_2 \lor \exists t_1 \exists t_2\, t = t_1 \times^\bullet t_2.$$

This is called the principle of *structural case analysis on the constructors of the numeric term t.*

We can use the above principle of structural case analysis in order to establish the admissibility of a certain kind of conditional discriminations on the constructors of numeric terms. These are of the form

$$\begin{aligned}&\mathit{Term}(t) \rightarrow \mathbf{case}\\&\qquad t = \mathrm{x}_i^\bullet \Rightarrow_i \beta_1[i]\\&\qquad t = n^\bullet \Rightarrow_n \beta_2[n]\\&\qquad t = t_1 +^\bullet t_2 \Rightarrow_{t_1,t_2} \beta_3[t_1,t_2]\\&\qquad t = t_1 \times^\bullet t_2 \Rightarrow_{t_1,t_2} \beta_4[t_1,t_2].\\&\quad \mathbf{end}\end{aligned}$$

Because of the precondition $\mathit{Term}(t)$, we have to evaluate only four alternatives instead of five. Moreover, characteristic terms of each alternative can be selected much simpler than those in Par. 9.1.2. For instance, we have

$$\mathit{Term}(t) \rightarrow \exists t_1 \exists t_2\, t = t_1 +^\bullet t_2 \leftrightarrow [t]_1^3 = 2$$

and thus we can use the expression $[t]_1^3 =_* 2$ as the characteristic term of the third variant of the above conditional. Compare with the characteristic term $\mathit{Tuple}_*(3,t) \wedge_* [t]_1^3 =_* 2$ of the same variant from Par. 9.1.2.

9.1.5 Structural induction on numeric terms. The principle of structural induction over numeric terms can be informally stated as follows. To prove by structural induction that a property $\varphi[t]$ holds for every numeric term t it suffices to prove:

Base cases: the property holds for every variable $\mathrm{x}_i^\bullet$ and constant $n^\bullet$.
Induction steps: if the property holds for the terms t_1, t_2 then it holds also for the terms $t_1 +^\bullet t_2$ and $t_1 \times^\bullet t_2$.

This is expressed formally in PA by

$$\begin{aligned}\vdash_{\mathrm{PA}} &\forall i\, \varphi[\mathrm{x}_i^\bullet] \wedge \forall n\, \varphi[n^\bullet] \wedge \forall t_1 \forall t_2 (\varphi[t_1] \wedge \varphi[t_2] \rightarrow \varphi[t_1 +^\bullet t_2]) \wedge\\&\forall t_1 \forall t_2 (\varphi[t_1] \wedge \varphi[t_2] \rightarrow \varphi[t_1 \times^\bullet t_2]) \rightarrow \mathit{Term}(t) \rightarrow \varphi[t]\end{aligned}$$

The theorem is called the principle of *structural induction on the numeric term t for* $\varphi[t]$.

Proof. The principle of structural induction for numeric terms is derived in PA as follows. Under the assumptions corresponding to the base cases and induction steps of the structural induction take any numeric term t and prove that $\varphi[t]$ holds by complete induction on t. We consider the following four cases according to Par. 9.1.4. The cases when $t = \mathrm{x}_i^\bullet$ or $t = n^\bullet$ are trivial. In the case when $t = t_1 +^\bullet t_2$ for some t_1, t_2 we have $\varphi[t_1]$ and $\varphi[t_2]$ by IH since $t_1 < t_1 +^\bullet t_2$ and $t_2 < t_1 +^\bullet t_2$. From the assumption we get $\varphi[t_1 +^\bullet t_2]$. The case when $t = t_1 \times^\bullet t_2$ for some t_1, t_2 is similar. □

9.1.6 Structural recursion on numeric terms. Structural induction over numeric terms is used to prove properties of functions defined by the scheme of *structural recursion on numeric terms*. In its simplest form, the operator of structural recursion over numeric terms introduces a function f from functions g_1, g_2, g_3 and g_4 satisfying

$$\begin{aligned} f(t,y) = \mathbf{case}\ & \\ & t = \mathrm{x}_i^\bullet \Rightarrow g_1(i,y) \\ & t = n^\bullet \Rightarrow g_2(n,y) \\ & t = t_1 +^\bullet t_2 \Rightarrow g_3\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & t = t_1 \times^\bullet t_2 \Rightarrow g_4\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & \mathbf{otherwise} \Rightarrow 0 \\ & \mathbf{end}. \end{aligned}$$

Note that this is a recursive definition regular in the first argument with discrimination on the constructors of numeric terms (output variables of each variant are omitted).

The following identities form the clausal form of the above definition

$$\begin{aligned} & f(\mathrm{x}_i^\bullet, y) = g_1(i,y) \\ & f(n^\bullet, y) = g_2(n,y) \\ & f(t_1 +^\bullet t_2, y) = g_3\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & f(t_1 \times^\bullet t_2, y) = g_4\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \end{aligned}$$

Note here that this is a typical example where we wish to use the default clauses – in this case

$$f(t,y) = 0 \leftarrow \neg\exists i\, t = \mathrm{x}_i^\bullet \land \neg\exists n\, t = n^\bullet \land \neg\exists t_1 \exists t_2\, t = t_1 +^\bullet t_2 \land \neg\exists t_1 \exists t_2\, t = t_1 \times^\bullet t_2$$

in order not to clutter the definition. We do not care what value is yielded by the application $f(t,y)$ if t is not the code of a numeric term.

The above definition for the function f can be easily rewritten to a conditional program for the same function as we have

$$\begin{aligned} \vdash_{\mathrm{PA}}\ Term(t) \rightarrow f(t,y) = \mathbf{case}\ & \\ & t = \mathrm{x}_i^\bullet \Rightarrow g_1(i,y) \\ & t = n^\bullet \Rightarrow g_2(n,y) \\ & t = t_1 +^\bullet t_2 \Rightarrow g_3\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & t = t_1 \times^\bullet t_2 \Rightarrow g_4\big(t_1, t_2, f(t_1,y), f(t_2,y), y\big) \\ & \mathbf{end} \end{aligned}$$

Its conditions of regularity, e.g. for the variant $t = t_1 +^\bullet t_2$

$$\begin{aligned} & \vdash_{\mathrm{PA}}\ Term(t) \land t = t_1 +^\bullet t_2 \rightarrow t_1 < t \land Term(t_1) \\ & \vdash_{\mathrm{PA}}\ Term(t) \land t = t_1 +^\bullet t_2 \rightarrow t_2 < t \land Term(t_2), \end{aligned}$$

are trivially satisfied.

Similar schemes, when we allow terms with arbitrary number of parameters on the right-hand side of the above identities, substitution in parameters, or even nested recursive applications, will be also called definitions by structural recursion on numeric terms.

9.1.7 Size of numeric terms. The function $|t|$ yields the *size* of the numeric term t, i.e. the number of operations including variables needed to construct the term t. The function is defined by parameterless structural recursion on the numeric term t as a p.r. function:

$|\mathrm{x}_i^\bullet| = 1$
$|n^\bullet| = 1$
$|t_1 +^\bullet t_2| = |t_1| + |t_2| + 1$
$|t_1 \times^\bullet t_2| = |t_1| + |t_2| + 1.$

9.1.8 Denotation of numeric terms. We now define the binary *denotation (valuation)* function $[\![t]\!]_v$ which takes the code t of a numeric term τ and the assignment v which is a list assigning the value $v[i]$ to the variable x_i and yields the value of the term τ. The function $[\![t]\!]_v$ is defined by structural recursion on the numeric term t as a p.r. function:

$[\![\mathrm{x}_i^\bullet]\!]_v = v[i]$
$[\![n^\bullet]\!]_v = n$
$[\![t_1 +^\bullet t_2]\!]_v = [\![t_1]\!]_v + [\![t_2]\!]_v$
$[\![t_1 \times^\bullet t_2]\!]_v = [\![t_1]\!]_v \times [\![t_2]\!]_v.$

For instance, if $v = \langle 10, 11, 12, 13, 0\rangle$ then

$$\begin{aligned}\left[\!\!\left[(\mathrm{x}_1^\bullet +^\bullet 2^\bullet) \times^\bullet \mathrm{x}_3^\bullet\right]\!\!\right]_v &= [\![\mathrm{x}_1^\bullet +^\bullet 2^\bullet]\!]_v \times [\![\mathrm{x}_3^\bullet]\!]_v = \big([\![\mathrm{x}_1^\bullet]\!]_v + [\![2^\bullet]\!]_v\big) \times [\![\mathrm{x}_3^\bullet]\!]_v = \\ &= \big(v[1] + 2\big) \times v[3] = (11+2) \times 13 = 169.\end{aligned}$$

9.1.9 The compiler and postfix machine. In this example we give the proof of correctness of a simple compiler for numeric terms. A term is compiled into a program of a postfix machine and then the program is executed.

The instructions are defined with the help of four pair constructors:

$$\begin{aligned}\mathit{LOAD}(i) &= \langle 0, i\rangle \\ \mathit{PUSH}(n) &= \langle 1, n\rangle \\ \mathit{ADD} &= \langle 2, 0\rangle \\ \mathit{MULT} &= \langle 3, 0\rangle.\end{aligned}$$

A program of the machine is just a list of instructions.

Numeric terms are compiled into programs with the help of $\mathit{Cmp}(t)$. The compilation function is defined by structural recursion on numeric terms as a p.r. function:

$\mathit{Cmp}(\mathrm{x}_i^\bullet) = \langle \mathit{LOAD}(i), 0\rangle$
$\mathit{Cmp}(n^\bullet) = \langle \mathit{PUSH}(n), 0\rangle$
$\mathit{Cmp}(t_1 +^\bullet t_2) = \mathit{Cmp}(t_1) \oplus \mathit{Cmp}(t_2) \oplus \langle \mathit{ADD}, 0\rangle$

$$Cmp(t_1 \times^\bullet t_2) = Cmp(t_1) \oplus Cmp(t_2) \oplus \langle MULT, 0 \rangle.$$

For instance, the following is the compiled program

$$\langle LOAD(1), PUSH(2), ADD, LOAD(3), MULT, 0 \rangle$$

for (the code of) the numeric term $(x_1 + 2) \times x_3$.

The operation of the postfix machine itself is described by the ternary function $Run(p, v, s)$, where p is a program, v is an assignment (environment), and s is a list of values (I/O stack). The function $Run(p, v, s)$ is defined by recursion on the list p with substitution in the parameter s as a p.r. function:

$$\begin{aligned}
&Run(0, v, \langle t, s \rangle) = t \\
&Run(\langle LOAD(i), p \rangle, v, s) = Run(p, v, \langle v[i], s \rangle) \\
&Run(\langle PUSH(n), p \rangle, v, s) = Run(p, v, \langle n, s \rangle) \\
&Run(\langle ADD, p \rangle, v, \langle t_2, t_1, s \rangle) = Run(p, v, \langle t_1 + t_2, s \rangle) \\
&Run(\langle MULT, p \rangle, v, \langle t_2, t_1, s \rangle) = Run(p, v, \langle t_1 \times t_2, s \rangle).
\end{aligned}$$

Note that the other parameter v does not change in recursion.

Correctness of the compiler is expressed by the following formula:

$$\vdash_{\mathrm{PA}} \ Term(t) \to Run(Cmp(t), v, 0) = [\![t]\!]_v \,. \tag{1}$$

In order to prove it we need the following auxiliary claim:

$$\vdash_{\mathrm{PA}} \ Term(t) \to \forall p \forall s \big(Run(Cmp(t) \oplus p, v, s) = Run(p, v, \langle [\![t]\!]_v, s \rangle) \big). \tag{2}$$

This is proved by structural induction on the numeric term t. So take any numbers p, s and continue by case analysis on the numeric term t. If $t = \mathrm{x}_i^\bullet$ for some i then we have

$$\begin{aligned}
Run(Cmp(\mathrm{x}_i^\bullet) \oplus p, v, s) &= Run(\langle LOAD(i), p \rangle, v, s) = Run(p, v, \langle v[i], s \rangle) = \\
&= Run(p, v, \langle [\![\mathrm{x}_i^\bullet]\!]_v, s \rangle).
\end{aligned}$$

If $t = t_1 +^\bullet t_2$ for some t_1, t_2 then we obtain

$$\begin{aligned}
&Run(Cmp(t_1 +^\bullet t_2) \oplus p, v, s) = \\
&\quad = Run\big(\langle Cmp(t_1) \oplus Cmp(t_2) \oplus \langle ADD, p \rangle \rangle, v, s\big) \overset{\mathrm{IH}}{=} \\
&\quad = Run\big(\langle Cmp(t_2) \oplus \langle ADD, p \rangle \rangle, v, \langle [\![t_1]\!]_v, s \rangle\big) \overset{\mathrm{IH}}{=} \\
&\quad = Run(\langle ADD, p \rangle, v, \langle [\![t_2]\!]_v, [\![t_1]\!]_v, s \rangle) = \\
&\quad = Run(p, v, \langle [\![t_1]\!]_v + [\![t_2]\!]_v, s \rangle) = Run(p, v, \langle [\![t_1 +^\bullet t_2]\!]_v, s \rangle).
\end{aligned}$$

Note that the first induction hypothesis is applied with $Cmp(t_2) \oplus \langle ADD, p \rangle$ in place of p while s is unchanged; and that the second induction hypothesis is applied with $\langle ADD, p \rangle$ and $\langle [\![t_1]\!]_v, s \rangle$ in place of p and s, respectively. The remaining cases are proved similarly.

We are now in position to prove (1). Take any term t and we have

$$Run(Cmp(t), v, 0) \stackrel{(2)}{=} Run(0, v, \langle [\![t]\!]_v, 0\rangle) = [\![t]\!]_v .$$

9.1.10 Rearranging terms into expressions with left associated addition. In this paragraph we give an example of a program which goes beyond structural recursion. Consider the problem of rearranging numeric terms so that the additions which they contain are associated to left. For instance, the term $(x_1 + x_2) + (x_3 + (x_4 + x_5))$ is transformed to an equivalent term $(((x_1 + x_2) + x_3) + x_4) + x_5$ with left associated addition.

More formally, let $Lassoc(t)$ be a predicate holding of terms with left associated addition. The predicate is defined by course of values recursion as primitive recursive by

$$\begin{aligned}
&Lassoc(t) \leftarrow \neg\exists t_1, t_2\, t = t_1 +^\bullet t_2 \\
&Lassoc(t_1 +^\bullet t_2) \leftarrow \neg\exists t_3, t_4\, t_2 = t_3 +^\bullet t_4 \wedge Lassoc(t_1).
\end{aligned}$$

We are looking for a p.r. function $f(t)$ satisfying

$$\vdash_{\mathrm{PA}} \; Term(t) \rightarrow Term\, f(t) \tag{1}$$
$$\vdash_{\mathrm{PA}} \; Term(t) \rightarrow Lassoc\, f(t) \tag{2}$$
$$\vdash_{\mathrm{PA}} \; Term(t) \rightarrow |f(t)| = |t| \tag{3}$$
$$\vdash_{\mathrm{PA}} \; Term(t) \rightarrow [\![f(t)]\!]_v = [\![t]\!]_v . \tag{4}$$

The desired function is defined by

$$\begin{aligned}
&f(t) = t \leftarrow \neg\exists t_1, t_2\, t = t_1 +^\bullet t_2 \\
&f(t_1 +^\bullet t_2) = f(t_1) +^\bullet t_2 \leftarrow \neg\exists t_3, t_4\, t_2 = t_3 +^\bullet t_4 \\
&f\big(t_1 +^\bullet (t_2 +^\bullet t_3)\big) = f(t_1 +^\bullet t_2 +^\bullet t_3).
\end{aligned}$$

Is this a correct definition? The first two clauses are structurally recursive, but this does not hold for the third, in which the recursion goes from $t_1 +^\bullet (t_2 +^\bullet t_3)$ to $t_1 +^\bullet t_2 +^\bullet t_3$. We claim that the above definition is the definition with measure $m(t)$:

$$\begin{aligned}
&m(t) = 1 \leftarrow \neg\exists t_1, t_2\, t = t_1 +^\bullet t_2 \\
&m(t_1 +^\bullet t_2) = m(t_1) + 2m(t_2) + 1.
\end{aligned}$$

Indeed, the regularity condition for the third clause follows from:

$$\begin{aligned}
m(t_1 +^\bullet t_2 +^\bullet t_3) &= m(t_1) + 2m(t_2) + 2m(t_3) + 2 < \\
&< m(t_1) + 2m(t_2) + 4m(t_3) + 3 = m(t_1 +^\bullet (t_2 +^\bullet t_3)).
\end{aligned}$$

Properties (1)-(4) can be proved straightforwardly by the corresponding induction principle.