Chapter 15

Recursive types

We have already seen a number of basic ways of making types for use in variable declarations. As well as the 'standard' types like the natural numbers $\mathbb{N}$ and the reals $\mathbb{R}$, we encountered the type-constructing functions **set**, **bag** and **seq** that make new types from existing ones, and we considered functional and relational types.

In this chapter we go further, showing how to invent such structuring tools for ourselves.

15.1 Disjoint union

The principal new ingredients of our type constructions will be disjoint union and recursion. Normal set union forms a new set from two given sets by including in the new set exactly those elements that are in either of the two given sets: thus for sets A, B and elements x, we have the equivalence $x \in A \cup B \equiv x \in A \vee x \in B$. One property that normal set union does *not* have in general, however, is equality between $\#(A \cup B)$ and $\#A + \#B$: that property holds only when A and B are disjoint. (Recall that the size of the (finite) type S is written $\#S$.)

Disjoint union resembles ordinary union in that it aggregates the elements of its two operands; but it makes them disjoint by giving each operand a distinct 'tag'. Thus, while the elements of the (ordinary) union $\{0, 1\} \cup \{1, 2\}$ are 0, 1 and 2, the elements of the disjoint union

$$\mathsf{left}\ \{0, 1\} \mid \mathsf{right}\ \{1, 2\}$$

are $\mathsf{left}\, 0$, $\mathsf{left}\, 1$, $\mathsf{right}\, 1$ and $\mathsf{right}\, 2$. The names left and right are the *tags*, made up by the programmer (or mathematician) specifically to keep the two sides disjoint: elements $\mathsf{left}\, 1$ and $\mathsf{right}\, 1$ are not equal.

In fact left and right are *injection functions*, because they inject the component types ($\{0, 1\}$ and $\{1, 2\}$) into the disjoint union type (and because they are injective). If we let the disjoint union above be X then we have the following function types for the injections:

$$\begin{array}{ll} \mathsf{left}: & \{0,1\} \rightarrow X \\ \mathsf{right}: & \{1,2\} \rightarrow X\ . \end{array}$$

Notice that X has four elements (not just three); and more generally, for any (finite) sets A and B we have

$$\#(\mathsf{left}\ A \mid \mathsf{right}\ B) \quad = \quad \#A + \#B\ .$$

If we write several types following a tag, rather than just one, then the injection functions take values of those types successively.[1] Thus for example

$$\mathsf{nats}\ \mathbb{N}\ \mathbb{N} \mid \mathsf{reals}\ \mathbb{R}\ \mathbb{R} \tag{15.1}$$

is a type including the elements $\mathsf{nats}\, 2\ 3$ and $\mathsf{reals}\, 1.5\ \pi$. (Writing $\mathsf{nats}\ (\mathbb{N} \times \mathbb{N}) \mid \mathsf{reals}\ (\mathbb{R} \times \mathbb{R})$ would have had a similar effect, but there the elements would have been $\mathsf{nats}(2,3)$ and $\mathsf{reals}(1.5,\pi)$ instead.)

It is also possible to have a tag just on its own, in which case just a single element, of that name, is contributed to the type. (Think of the injection function in that case as a constant, which is after all just a function of no arguments.)

Here are some examples of disjoint unions:

1. The type of days of the week is

   $$\mathsf{sunday} \mid \mathsf{monday} \mid \mathsf{tuesday} \mid \mathsf{wednesday} \mid \mathsf{thursday} \mid \mathsf{friday} \mid \mathsf{saturday}\ .$$

   Typical elements are sunday and monday.
2. A value that is either an integer or is 'undefined' could be typed

   $$\mathsf{ok}\ \mathbb{Z} \mid \mathsf{undefined}\ .$$

   Typical values are $\mathsf{ok}\, 3$ and $\mathsf{undefined}$.
3. A collection of currencies could be typed

   $$\mathsf{pounds}\ \mathbb{Z} \mid \mathsf{guilders}\ \mathbb{Z} \mid \mathsf{ecu}\ \mathbb{Z} \mid \mathsf{lira}\ \mathbb{Z}\ .$$

   Typical values would be $\mathsf{pounds}\, 3$ and $\mathsf{lira}(-1000000)$.
4. A database entry might either be empty or associate a key of type K with a datum of type D; its type would be

   $$\mathsf{empty} \mid \mathsf{full}\ K\ D\ .$$

   Typical values are empty and $\mathsf{full}\, k\ d$, with k of type K and d of type D.

Note that the injections are written in the function font, just as other functions and predicates are.

[1] That is, the injection functions are *Curried.*

15.2 Testing for tags

Consider the type $\mathsf{empty} \mid \mathsf{full}\, K\ D$ in the final example above. Given an element x of it, we may wish to know which alternative it inhabits ('what its tag is'). Although the formulae

$$\begin{array}{rl} & x = \mathsf{empty} \\ \text{and} & x = \mathsf{full}\, k\ d\ , \end{array}$$

make sense for $k : K$ and $d : D$, they are not exactly what we want. The first formula indeed determines whether x is in the first alternative. But the second is too specific, depending on a particular k and d: instead we need the existential quantification $(\exists\, k : K;\ d : D \cdot x = \mathsf{full}\, k\ d)$, asking whether $x = \mathsf{full}\, k\ d$ for *some* k and d.

We therefore introduce the abbreviation

$$x \ \mathsf{is}\ \mathsf{full},$$

for $(\exists\, k : K;\ d : D \cdot x = \mathsf{full}\, k\ d)$, using a new two-place predicate is for the purpose: it tests for membership of an alternative.

15.3 Pattern-matching in alternations

Testing as in the previous section is not yet enough to make our disjoint unions useful: the truth of $m\ \mathsf{is}\ \mathsf{guilders}$ tells us what kind of money we have, but not how much. Consider for example the type definition

$$\mathsf{first} \mid \mathsf{second}\ A \mid \mathsf{third}\ B\ C\ ,$$

and suppose E is some expression of that type; then the program

$$\begin{array}{l} \textbf{if}\ E\ \textbf{is} \\ \quad \mathsf{first} \qquad \rightarrow prog1 \\ [\!]\ \ \mathsf{second}\ a \rightarrow prog2 \\ [\!]\ \ \mathsf{third}\ b\ c \rightarrow prog3 \\ \textbf{fi} \end{array}$$

executes $prog1$, $prog2$, or $prog3$ depending on the the tag of E. In addition, the variables a, b, and c are local in their respective branches of the alternation, and are initialised appropriately. Thus in general we access the components of elements of such types with a *tagged alternation*, as above.

Note that (sanserif) is (of the previous section) is a binary predicate, while (bold) **is** simply distinguishes the tagged alternation from the ordinary one. Although the effect of the tagged alternation can be partially achieved using the predicate is instead, the expression E in that case must be repeated, and in $prog2'$, for example, the variable a is not available as it is in $prog2$:

$$\begin{array}{l} \textbf{if}\ E\ \mathsf{is\ first} \rightarrow prog1' \\ [\!]\ \ E\ \mathsf{is\ second} \rightarrow prog2' \\ [\!]\ \ E\ \mathsf{is\ third} \rightarrow prog3' \\ \textbf{fi} \end{array}$$

is less useful. That shows the advantage of the pattern matching.

If the expression E does not match any of the supplied patterns in a tagged alternation, then the alternation aborts (just as ordinary alternation aborts when no guard is true).

Although the use of tagged alternation should not need much formal justification in practice, it does — like the other constructs of our language — have an associated refinement law.

Law 15.1 tagged alternation Let first, middle and last be tags from a typical type declaration

$$\mathsf{first}\ A \cdots H \mid \mathsf{middle}\ I \cdots P \mid \mathsf{last}\ Q \cdots Z\ .$$

Provided none of $a \cdots h, q \cdots z$ appear free in E or *prog*, this refinement is valid:

$$\begin{array}{ll} & \{E\ \mathsf{is\ first} \vee E\ \mathsf{is\ last}\}\ prog \\ \sqsubseteq & \textbf{if}\ E\ \textbf{is} \\ & \quad \mathsf{first}\ a \cdots h \rightarrow \{E = \mathsf{first}\ a \cdots h\}\ prog \\ & [\!]\ \ \mathsf{last}\ q \cdots z \rightarrow \{E = \mathsf{last}\ q \cdots z\}\ prog \\ & \textbf{fi}\ . \end{array}$$

□

We have of necessity given just an example of the types to which law *tagged alternation* 15.1 applies, but the principle should be clear: the assumption $\{E\ \mathsf{is\ first} \vee E\ \mathsf{is\ last}\}$ makes explicit the possible tags of E — which need not be all that the type allows. (We have left out middle, for example.) The guards of the alternation then correspond exactly to the disjuncts of the assumption.

Here are some alternations based on the example types mentioned earlier:

1. $$\begin{array}{l} \textbf{if}\ dd\ \textbf{is} \\ \quad \mathsf{wednesday} \rightarrow prog1 \\ [\!]\ \ \mathsf{tuesday} \rightarrow prog2 \\ \textbf{fi} \end{array}$$
   executes $prog1$ if dd is wednesday holds, and $prog2$ if dd is tuesday holds; it aborts otherwise.
2. $$\begin{array}{l} \textbf{if}\ nn\ \textbf{is} \\ \quad \mathsf{undefined} \rightarrow \textbf{skip} \\ [\!]\ \ \mathsf{ok}\ n \rightarrow nn := \mathsf{ok}(n+1) \\ \textbf{fi} \end{array}$$
   'increments' nn if it is not undefined; if undefined it is left so.

3. $$\begin{array}{l}\textbf{if } mm \textbf{ is} \\ \quad \mathsf{guilders}\, m \rightarrow mm := \mathsf{ecu}(.45 \times m) \\ [\!]\ \mathsf{pounds}\, m \rightarrow mm := \mathsf{ecu}(1.3 \times m) \\ [\!]\ \mathsf{lira}\, m \quad\ \rightarrow mm := \mathsf{ecu}(m/1885) \\ [\!]\ \mathsf{ecu}\, m \quad\ \rightarrow \textbf{skip} \\ \textbf{fi}\end{array}$$
converts m into European Currency Units. (The ecu branch does nothing; but what would happen if it were not there?)
4. Suppose we represent a database db as a sequence of length N of entries, each either empty or containing a key/datum pair:

$$db : \mathbf{seq}_N(\mathsf{empty} \mid \mathsf{full}\ K\ \ D)\ .$$

Then for given $k : K$ the following program searches the database for the associated datum d in D:

$$\begin{array}{l} i := 0; \\ \textbf{re } S\bullet \\ \quad \textbf{if } db[i] \textbf{ is} \\ \quad\quad \mathsf{empty} \rightarrow i := i+1;\ S \\ \quad [\!]\ \mathsf{full}\, k'\ \ d' \rightarrow \\ \quad\quad\quad \textbf{if } k = k' \rightarrow d := d' \\ \quad\quad\quad [\!]\ \ k \neq k' \rightarrow i := i+1;\ S \\ \quad\quad\quad \textbf{fi} \\ \quad \textbf{fi} \\ \textbf{er}\ . \end{array}$$

Note how in the pattern-matching guards the local variables are k' and d', allowing a distinction to be made between the values found in the sequence and the original k and d. The program behaves unpredictably if the key does not occur in the database.

As an example of the use of *tagged alternation* 15.1, we will now look at the development steps that led to the last of the four program fragments above. First we define for convenience a predicate expressing the presence (or absence) of k in db:

$$k \ \mathsf{in}\ db \ \ \widehat{=}\ \ (\exists\, d \bullet \mathsf{full}\, k\ \ d \in db)\ .$$

Then, starting with a specification that allows abortion (hence, unpredictable behaviour) if k does not occur, we proceed:

$$\begin{array}{ll} & d\colon [k \ \mathsf{in}\ db\ ,\ \mathsf{full}\, k\ \ d \in db] \\ \sqsubseteq & \textbf{var } i : \mathbb{N}\bullet \\ & i := 0; \\ & d, i\colon [k \ \mathsf{in}\ db{\downarrow}i\ ,\ \mathsf{full}\, k\ \ d \in db] \qquad\qquad \triangleleft \end{array}$$

$$\begin{array}{lll}
& \sqsubseteq & \textbf{re } S \textbf{ variant } I \textbf{ is } i \bullet \\
& & d, i\colon [I = i \wedge k \text{ in } db{\downarrow}i\ ,\ \mathsf{full}\,k\ d \in db] \quad \triangleleft \\
& \sqsubseteq & \text{“}k \text{ in } db{\downarrow}i \ \Rrightarrow\ i < N \ \Rrightarrow\ db[i] \text{ is empty} \vee db[i] \text{ is full”} \\
& & \textbf{if } db[i] \textbf{ is} \\
& & \quad \mathsf{empty} \to d, i\colon \left[\begin{array}{l} db[i] = \mathsf{empty} \\ I = i < N \\ k \text{ in } db{\downarrow}i \end{array} ,\ \mathsf{full}\,k\ d \in db \right] \qquad \text{(i)} \\
& & [\!]\ \ \mathsf{full}\,k'\ d' \to d, i\colon \left[\begin{array}{l} db[i] = \mathsf{full}\,k'\ d' \\ I = i < N \\ k \text{ in } db{\downarrow}i \end{array} ,\ \mathsf{full}\,k\ d \in db \right] \qquad \text{(ii)} \\
& & \textbf{fi} \\
\text{(i)} & \sqsubseteq & i := i + 1; \\
& & d, i\colon [I < i \le N \wedge k \text{ in } db{\downarrow}i\ ,\ \mathsf{full}\,k\ d \in db] \quad \triangleleft \\
& \sqsubseteq & S \\
\text{(ii)} & \sqsubseteq & \textbf{if } k = k' \to d, i\colon \left[\begin{array}{l} i < N \\ db[i] = \mathsf{full}\,k\ d' \end{array} ,\ \mathsf{full}\,k\ d \in db \right] \qquad \text{(iii)} \\
& & [\!]\ \ k \ne k' \to d, i\colon \left[\begin{array}{l} k \ne k' \\ db[i] = \mathsf{full}\,k'\ d' \\ I = i < N \\ k \text{ in } db{\downarrow}i \end{array} ,\ \mathsf{full}\,k\ d \in db \right] \qquad \text{(iv)} \\
& & \textbf{fi} \\
\text{(iii)} & \sqsubseteq & d := d' \\
\text{(iv)} & \sqsubseteq & \text{“as for (i)”} \\
& & i := i + 1; \\
& & S\ .
\end{array}$$

The variant I is increasing, in this example: it starts at i and is bounded above by N. (Following the earlier presentation strictly, our variant would have been $N - i$, bounded below by 0.)

15.4 Type declarations

With the introduction of our new type-forming mechanisms it becomes more important that we allow types to be named and reused. The syntax is, by analogy with procedures,

$$\textbf{type } \mathit{Name} \ \widehat{=}\ \mathit{type{-}expression}\ ,$$

thus allowing declarations like

$$\begin{array}{lll} \textbf{type}\ \mathit{Money} & \mathrel{\widehat{=}} & \mathsf{pounds}\ \mathbb{Z} \mid \mathsf{guilders}\ \mathbb{Z} \mid \mathsf{ecu}\ \mathbb{Z} \mid \mathsf{lira}\ \mathbb{Z} \\ \quad \mathit{Days} & \mathrel{\widehat{=}} & \mathsf{sunday} \mid \mathsf{monday} \mid \mathsf{tuesday} \\ & & \mid \mathsf{wednesday} \mid \mathsf{thursday} \mid \mathsf{friday} \mid \mathsf{saturday} \\ \quad \mathit{Entry} & \mathrel{\widehat{=}} & \mathsf{empty} \mid \mathsf{full}\ K\ D \\ \quad \mathit{DataBase} & \mathrel{\widehat{=}} & \mathbf{seq}_N\ \mathit{Entry}\ . \end{array}$$

Unless the types refer to each other recursively (a case we treat shortly), the meanings of programs with such declarations can be recovered simply by replacing type names by the expressions they stand for. Thus the declaration

$$\textbf{var}\ db : \mathit{DataBase}$$

means

$$\textbf{var}\ db : \mathbf{seq}_N\ \mathit{Entry}$$

which in turn means

$$\textbf{var}\ db : \mathbf{seq}_N(\mathsf{empty} \mid \mathsf{full}\ K\ D)\ .$$

We allow type parameters as well, writing for example

$$\mathit{Maybe}\ T \mathrel{\widehat{=}} \mathsf{ok}\ T \mid \mathsf{undefined}\ ,$$

and, again in the absence of recursion, the meaning is recovered by substitution:

$$\begin{array}{ll} & \textbf{var}\ \mathit{marks} : \mathbf{seq}_N\ \mathit{Maybe}\ \mathbb{N} \\ \text{means} & \textbf{var}\ \mathit{marks} : \mathbf{seq}_N(\mathsf{ok}\ \mathbb{N} \mid \mathsf{undefined})\ . \end{array}$$

15.5 Recursive types

We are now well-enough equipped to discuss type declarations that refer to themselves. Consider, as our first example, the recursive declaration

$$\textbf{type}\ \mathit{Tree} \mathrel{\widehat{=}} \mathsf{empty} \mid \mathsf{node}\ \mathbb{N}\ \mathit{Tree}\ \mathit{Tree}\ . \tag{15.2}$$

Typical elements of the type are

$$\begin{array}{l} \mathsf{empty} \\ \mathsf{node}\,3\ \mathsf{empty}\ \mathsf{empty} \\ \mathsf{node}\,7\ (\mathsf{node}\,3\ \mathsf{empty}\ \mathsf{empty})\ \mathsf{empty}\ . \end{array}$$

One might depict them as in Figure 15.1, where their tree-like nature is evident.

In fact we define the elements of a recursive (or any other) type to be exactly those whose membership in the type can be demonstrated by (perhaps repeated, but only finitely often) reference to the type definition. In the case above, for example, we know that empty is a *Tree* from the left alternative of the declaration, and it does not matter what the right alternative says. Once we do know that empty

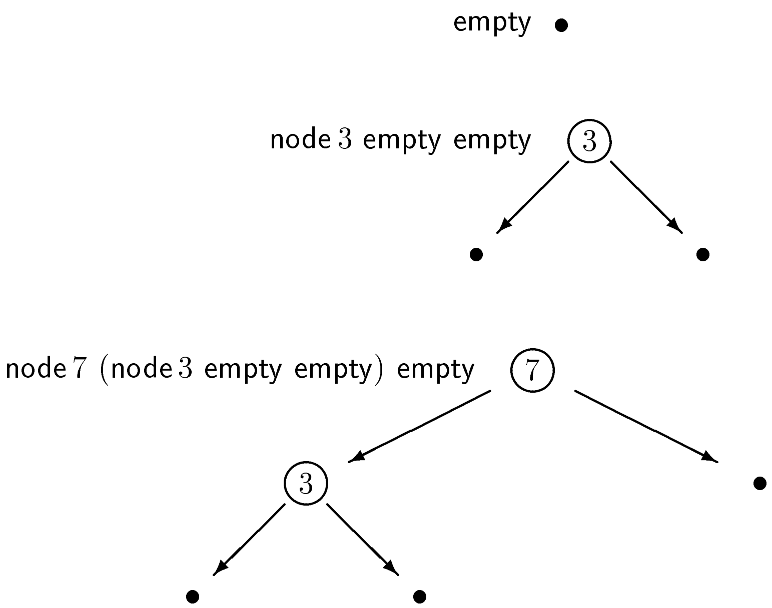

Figure 15.1 Elements of type *Tree*

is a *Tree*, we can use the right alternative to deduce that $\mathsf{node}\,3\ \mathsf{empty}\ \mathsf{empty}$ is as well (and the same holds of course for other natural numbers, not only 3). Once we know *that*, we consider $\mathsf{node}\,7\ (\mathsf{node}\,3\ \mathsf{empty}\ \mathsf{empty})\ \mathsf{empty}$, and so on.

The above is just about the simplest view one could take to recursively defined types, and it does restrict us to finite structures, excluding for example the 'infinite' tree of Figure 15.2: $\mathsf{node}\,0\ \mathsf{empty}\ (\mathsf{node}\,1\ \mathsf{empty}\ (\mathsf{node}\,2\ (\cdots)))$. There is nothing inherently wrong with such infinite structures, although we do not treat them here; but there can be problems however with declarations such as

$$\mathbf{type}\ \mathit{TooBig}\ \widehat{=}\ \mathsf{one} \mid \mathsf{several}\ (\mathbf{set}\ \mathit{TooBig})\ .$$

Since no set (or type) can be big enough to contain representations of all subsets of itself, it is not at all certain that the type *TooBig* exists, in spite of the fact that we can enumerate elements as before:

$$\begin{array}{l} \mathsf{one} \\ \mathsf{several}\{\} \\ \mathsf{several}\{\mathsf{one}\} \\ \mathsf{several}\{\mathsf{several}\{\}\} \\ \mathsf{several}\{\mathsf{one}, \mathsf{several}\{\}\}\ . \end{array}$$

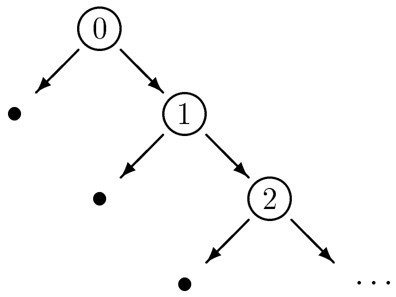

Figure 15.2 An 'infinite' tree

We avoid such problems by restricting our use of type functions within recursive definitions, avoiding **set** and **bag** in particular. (The function **seq** we allow, however.)

15.6 Structural order

Along with a definition such as (15.2) of *Tree* comes an ordering relation which we will write as $\olessthan$: it is the 'is-a-component-of' order, relating two elements whenever one occurs structurally within the other. For example,

$$
\begin{array}{ll}
 & \mathsf{empty} \\
\olessthan & \mathsf{node}\,3\ \mathsf{empty}\ \mathsf{empty} \\
\olessthan & \mathsf{node}\,7\ (\mathsf{node}\,3\ \mathsf{empty}\ \mathsf{empty})\ \mathsf{empty}\ .
\end{array}
$$

The order $\olessthan$ has these properties in common with the (strict) order $<$ over the integers, for example:

- It is *irreflexive*: for all x we have $\neg(x \olessthan x)$;
- It is *antisymmetric*: for all x and y we have $x \olessthan y \wedge y \olessthan x \Rightarrow \mathsf{false}$;
- It is *transitive*: for all x, y and z we have $x \olessthan y \wedge y \olessthan z \Rightarrow x \olessthan z$.

The order $<$ has one property that $\olessthan$ does not share, however: it is a *total* order, which means that for all distinct x and y we have that either $x < y$ or $y < x$. Because the same is not true for $\olessthan$, we call it a (strict) *partial* order, and strict partial orders are characterised exactly by the three properties above.

To see that $\olessthan$ is partial, consider

$$\mathsf{node}\,0\ \mathsf{empty}\ \mathsf{empty} \quad \text{and} \quad \mathsf{node}\,1\ \mathsf{empty}\ \mathsf{empty}\ .$$

Neither occurs as a component of the other, and so they are unrelated by $\olessthan$.

When developing programs over recursive data structures — as we shall do shortly — often the variant function is not an integer expression. Instead it is some element of the recursive type itself, and rather than use $<$, as we would over integers, we use the structural order $\circledless$ over the recursive type. For that to be sound, we use one additional property of $\circledless$ as a structural order, that it is *well-founded*: for no starting point x is there an infinite descending chain

$$\cdots \circledless x''' \circledless x'' \circledless x' \circledless x \; .$$

Note that our total order $<$ is well-founded over the natural numbers, since no natural number is negative. Thus our usual technique, writing $0 \leq v < v_0$ for integer-valued v, is a special case of the more general approach we are examining at the moment: the $0 \leq v$ restricts attention to the natural numbers (rather than the integers as a whole including negatives), and $<$ is well-founded over the natural numbers.

15.7 Pattern matching in iterations

By analogy with tagged alternations, we can define tagged iterations: where a tagged alternation would abort in the case of no match, a tagged iteration simply terminates ('successfully'). Consider for example the type

$$NatList \;\widehat{=}\; \mathsf{empty} \mid \mathsf{cons}\; \mathbb{N}\; NatList \; , \tag{15.3}$$

whose typical elements include

$$\begin{array}{l} \mathsf{empty} \\ \mathsf{cons}\, 0\; \mathsf{empty} \\ \mathsf{cons}\, 1\; (\mathsf{cons}\, 0\; \mathsf{empty}) \; . \end{array}$$

Given the declarations $n : \mathbb{N}$; $nl : NatList$, the following iteration assigns to n the sum of the elements originally in nl:

$$\begin{array}{l} n := 0; \\ \mathbf{do}\; nl\; \textsf{is}\; \mathsf{cons}\, n'\; nl' \rightarrow \\ \qquad n, nl := n + n', nl' \\ \mathbf{od}\; . \end{array}$$

(Note that nl is modified in the process.) The iteration guard covers only one of the two possible tags for nl, and so termination occurs when nl has the tag of the other alternative: that is, termination occurs when nl is empty.

The law for the introduction of tagged iterations is

Law 15.2 <u>tagged iteration</u> Let first, middle and last be tags from a type declaration

$$Type \;\widehat{=}\; \mathsf{first}\; A \cdots H \mid \mathsf{middle}\; I \cdots P \mid \mathsf{last}\; Q \cdots Y \; .$$

Provided none of $a \cdots h, q \cdots y$ appears free in z, inv, E, or V, this refinement is valid:

$$\begin{array}{ll}
& z\colon [inv\ ,\ inv \wedge \neg(E \text{ is first} \vee E \text{ is last})] \\
\sqsubseteq & \mathbf{do}\ E\ \mathbf{is} \\
& \quad \mathsf{first}\ a \cdots h \rightarrow z\colon [E = \mathsf{first}\ a \cdots h\ ,\ inv\ ,\ V \otimes V_0] \\
& [\!]\quad \mathsf{last}\ q \cdots y \rightarrow z\colon [E = \mathsf{last}\ q \cdots y\ ,\ inv\ ,\ V \otimes V_0] \\
& \mathbf{od}\ .
\end{array}$$

The formula *inv* is the invariant, the expression V is the variant, and the relation $\otimes$ must be well-founded.

□

Note that, as in tagged alternation, the guards of the iteration correspond exactly to the disjunction (negated) in the postcondition, and that they need not exhaust the type. (Indeed, if they did then the iteration would never terminate!)

As an example, we define the function $\mathsf{suml} : NatList \rightarrow \mathbb{N}$, for summing a list, as

$$\begin{array}{lcl}
\mathsf{suml}\,\mathsf{empty} & \widehat{=} & 0 \\
\mathsf{suml}(\mathsf{cons}\,n\ nl) & \widehat{=} & n + \mathsf{suml}\,nl\ ,
\end{array} \tag{15.4}$$

and develop the program above as follows:

$$\begin{array}{ll}
& n, nl := \mathsf{suml}\,nl, ? \\
\sqsubseteq & \mathbf{con}\ N : \mathbb{N}\bullet \\
& n, nl\colon [N = \mathsf{suml}\,nl\ ,\ n = N] \\
\sqsubseteq & I\ \widehat{=}\ N = n + \mathsf{suml}\,nl\bullet \\
& n := 0; \\
& n, nl\colon [I\ ,\ I \wedge nl \text{ is empty}] \qquad \triangleleft \\
\sqsubseteq & n, nl\colon [I\ ,\ I \wedge \neg(nl \text{ is cons})] \\
\sqsubseteq & \text{“\textit{tagged iteration} 15.2”} \\
& \mathbf{do}\ nl\ \mathbf{is}\ \mathsf{cons}\,n'\ nl' \rightarrow \\
& \quad n, nl\colon [nl = \mathsf{cons}\,n'\ nl'\ ,\ I\ ,\ nl \otimes nl_0] \qquad \triangleleft \\
& \mathbf{od} \\
\sqsubseteq & n, nl\colon \left[\begin{array}{l} nl' \otimes nl \\ N = n + n' + \mathsf{suml}\,nl' \end{array}\ ,\ N = n + \mathsf{suml}\,nl \wedge nl \otimes nl_0\right] \\
\sqsubseteq & n, nl := n + n', nl'\ .
\end{array}$$

Since the type *NatList* is structurally identical to $\mathbf{seq}\,\mathbb{N}$, we could almost have ‘defined’ the latter

$$\mathbf{seq}\,\mathbb{N}\ \widehat{=}\ \langle\rangle \mid \mathbb{N}{:}(\mathbf{seq}\,\mathbb{N})\ .$$

Continuing that analogy, if *nl* were declared of type $\mathbf{seq}\ \mathbb{N}$ we could have written for the above program

$$\begin{array}{l}
n := 0; \\
\mathbf{do}\ nl\ \mathbf{is}\ n'{:}nl' \rightarrow n, nl := n + n', nl'\ \mathbf{od}\ ,
\end{array}$$

which is a very compact imperative notation for summing a sequence.

15.8 Example: Summing a tree

We return now to our definition (15.2) of trees of ℕ, and we develop code to sum the node values. As one would expect, the recursive code is straightforward (type *Tree* is recursive, after all); iterative code will require some ingenuity, however. In this section we investigate both approaches.

15.8.1 Recursive tree-summing

First, we define sumt ('sum for trees' as opposed to lists)

$$\begin{array}{lcl} \mathsf{sumt\,empty} & = & 0 \\ \mathsf{sumt}(\mathsf{node}\,n\ \ nt1\ \ nt2) & = & n + \mathsf{sumt}\,nt1 + \mathsf{sumt}\,nt2\ , \end{array} \tag{15.5}$$

and then begin the development using a procedure:

$$\begin{array}{ll}
 & \mathbf{procedure}\ \mathit{SumT}\ (\mathbf{value}\ nt : \mathit{Tree};\ \ \mathbf{result}\ n : \mathbb{N}) \\
 & \widehat{=}\ \ n := \mathsf{sumt}\,nt \\
\sqsubseteq & \mathbf{variant}\ NT\ \mathbf{is}\ nt \bullet \\
 & n{:}\ [nt = NT\ ,\ n = \mathsf{sumt}\,nt] \\
\sqsubseteq & \mathbf{if}\ nt\ \mathbf{is} \\
 & \quad \mathsf{empty} \rightarrow n := 0 \\
 & [\!]\ \ \mathsf{node}\,n'\ nt1\ nt2 \rightarrow \\
 & \qquad n{:}\ [NT = nt = \mathsf{node}\,n'\ nt1\ nt2\ ,\ n = \mathsf{sumt}\,nt] \qquad \triangleleft \\
 & \mathbf{fi} \\
\sqsubseteq & n{:}\ \left[\begin{array}{l} nt1 \otimes NT \\ nt2 \otimes NT \end{array}\ ,\ n = n' + \mathsf{sumt}\,nt1 + \mathsf{sumt}\,nt2\right] \\
\sqsubseteq & \mathbf{var}\ n1, n2 : \mathbb{N} \\
 & n1{:}\ [nt1 \otimes NT\ ,\ n1 = \mathsf{sumt}\,nt1]\,; \qquad \text{(i)} \\
 & n2{:}\ [nt2 \otimes NT\ ,\ n2 = \mathsf{sumt}\,nt2]\,; \qquad \text{(ii)} \\
 & n := n' + n1 + n2 \\
\text{(i)}\ \sqsubseteq & \mathit{SumT}\ (nt1, n1) \\
\text{(ii)}\ \sqsubseteq & \mathit{SumT}\ (nt2, n2)\ .
\end{array}$$

The result is shown in Figure 15.3.

15.8.2 Iterative tree-summing

Matters are not so straightforward for the iterative development. Comparing the two types of sum (15.4) and (15.5), the first for lists and the second for trees, we find one recursive occurrence in the first case but two in the second. (That is, sumt 'calls itself' *twice* on the right-hand side.) Such multiple occurrences are not so easily dealt with by iteration.

$$
\begin{array}{l}
\textbf{procedure } SumT\ (\textbf{value } nt : Tree; \textbf{result } n : \mathbb{N}) \ \widehat{=} \\
|[\ \ \textbf{var } n1, n2 : \mathbb{N}\bullet \\
\quad \textbf{if } nt \textbf{ is} \\
\quad\quad \mathsf{empty} \rightarrow n := 0 \\
\quad [\!]\ \mathsf{node}\, n'\ nt1\ nt2 \rightarrow \\
\quad\quad\quad SumT\ (nt1, n1); \\
\quad\quad\quad SumT\ (nt2, n2); \\
\quad\quad\quad n := n' + n1 + n2 \\
\quad \textbf{fi} \\
]|
\end{array}
$$

Figure 15.3 Recursive tree-summing

Thus we look at replacing (15.5) by something more like (15.4). On the right-hand side of (15.5) we find

$$\mathsf{sumt}\, nt1 + \mathsf{sumt}\, nt2\ ,$$

and note that it can be rewritten

$$\mathsf{suml}\langle \mathsf{sumt}\, nt1, \mathsf{sumt}\, nt2\rangle\ .$$

We are generalising, in other words, and if we carry on for more than two we would define[2]

$$
\begin{array}{lcl}
\mathsf{sumlt}\langle\rangle & \widehat{=} & 0 \\
\mathsf{sumlt}(nt{:}ntl) & \widehat{=} & \mathsf{sumt}\, nt + \mathsf{sumlt}\, ntl\ .
\end{array} \tag{15.6}
$$

We now have only one recursive occurrence of sumlt, and we can obtain our original sumt as a special case of it.

Our iterative development is thus

$$
\begin{array}{ll}
& n := \mathsf{sumt}\, nt \\
\sqsubseteq & \textbf{var } ntl : \textbf{seq } Tree\bullet \\
& ntl := \langle nt\rangle; \\
& n, ntl := \mathsf{sumlt}\, ntl, ? \qquad\qquad \triangleleft \\
\sqsubseteq & \textbf{con } N : \mathbb{N}\bullet \\
& n, ntl\colon [N = \mathsf{sumlt}\, ntl\ ,\ n = N] \\
\sqsubseteq & I\ \widehat{=}\ N = n + \mathsf{sumlt}\, ntl\bullet \\
& n := 0; \\
& n, ntl\colon [I\ ,\ I \wedge ntl = \langle\rangle] \qquad\qquad \triangleleft
\end{array}
$$

[2]Functional programmers will recognise this as just $\mathsf{suml} \circ \mathsf{map}\,\mathsf{sumt}$.

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ ntl : \mathbf{seq}\ Tree \bullet \\
\quad n, ntl := 0, \langle nt \rangle; \\
\quad \mathbf{do}\ ntl\ \mathbf{is} \\
\qquad \mathsf{empty}{:}ntl' \rightarrow ntl := ntl' \\
\quad [\!]\ \ (\mathsf{node}\ n' nt1\ \ nt2){:}ntl' \rightarrow n, ntl := n + n', nt1{:}nt2{:}ntl' \\
\quad \mathbf{od} \\
]|
\end{array}
$$

Figure 15.4 Iterative tree-summing

$$
\begin{array}{lll}
& \sqsubseteq & \text{“invariant } I\text{; variant } V \text{ (later)”} \\
& & \mathbf{do}\ ntl\ \mathbf{is} \\
& & \qquad \mathsf{empty}{:}ntl' \rightarrow n, ntl{:}\ [ntl = \mathsf{empty}{:}ntl'\ ,\ I\ ,\ 0 \le V < V_0] \qquad \text{(i)} \\
& & [\!]\ \ (\mathsf{node}\ n'\ nt1\ \ nt2){:}ntl' \rightarrow \\
& & \qquad n, ntl{:}\ [ntl = (\mathsf{node}\ n'\ nt1\ \ nt2){:}ntl'\ ,\ I\ ,\ 0 \le V < V_0] \qquad \text{(ii)} \\
& & \mathbf{od} \\
\text{(i)} & \sqsubseteq & ntl := ntl' \\
\text{(ii)} & \sqsubseteq & n, ntl := n + n', nt1{:}nt2{:}ntl'\ . \qquad \text{(iii)}
\end{array}
$$

Those last two steps, perhaps rather large at first sight, are easily justified (still leaving aside the variant). For the second, we could have proceeded more slowly

$$
\begin{array}{lll}
\text{(ii)} & \sqsubseteq & n, ntl{:}\ [N = n + \mathsf{sumlt}((\mathsf{node}\ n'\ nt1\ \ nt2){:}ntl')\ , \\
& & \qquad N = n + \mathsf{sumlt}\ ntl \wedge 0 \le V < V_0] \\
& \sqsubseteq & n, ntl{:}\ [N = n + n' + \mathsf{sumt}\ nt1 + \mathsf{sumt}\ nt2 + \mathsf{sumlt}\ ntl'\ , \\
& & \qquad N = n + \mathsf{sumlt}\ ntl \wedge 0 \le V < V_0] \\
& \sqsubseteq & n, ntl{:}\ [N = n + n' + \mathsf{sumlt}(nt1{:}nt2{:}ntl')\ , \\
& & \qquad N = n + \mathsf{sumlt}\ ntl \wedge 0 \le V < V_0] \\
& \sqsubseteq & \text{“ignoring the variant part”} \\
& & \text{(iii) above.}
\end{array}
$$

The collected code for the iterative version is shown in Figure 15.4. The sequence *ntl* that occurs there is typical of imperative implementations of problems that are inherently recursive: examination of the code shows that *ntl* is accessed as a stack.

But that does leave the variant V. The 'obvious' choice of *ntl* itself (using the structural order for lists) does not work, since the second branch of the iteration actually *increases* the size of *ntl*, albeit with structurally smaller elements. What is decreased is in fact the *overall* size of ntl, taking its components into account:

$$\begin{array}{lcl} \mathsf{sizelt}\langle\rangle & \widehat{=} & 0 \\ \mathsf{sizelt}(nt{:}ntl) & \widehat{=} & \mathsf{sizet}\, nt + \mathsf{sizelt}\, ntl \end{array}$$

$$\begin{array}{lcl} \mathsf{sizet}\ \mathsf{empty} & \widehat{=} & 1 \\ \mathsf{sizet}(\mathsf{node}\, n\ \ nt1\ \ nt2) & \widehat{=} & 1 + \mathsf{sizet}\, nt1 + \mathsf{sizet}\, nt2\ . \end{array}$$

Thus our variant V is $\mathsf{sizelt}\, ntl$: with these more elaborate structures, variants are not as easily found as before!

15.9 Exercises

Ex. 15.1 What is the size of the type

$$\mathsf{first} \mid \mathsf{second}\ A \mid \mathsf{third}\ B\ \ C,$$

in terms of $\#A$, $\#B$, and $\#C$?

Ex. 15.2 ♡ Give types for each of the following:

1. A one-place buffer which either is empty or contains a natural number;
2. The finite binary trees with real numbers at the tips (a recursive type);
3. The colours of the spectrum.

Ex. 15.3 ♡ Example alternation 2 of Section 15.3 leaves n untouched if it is undefined. Write an alternation that instead *aborts* if n is 'undefined'.

Ex. 15.4 ♡ Recall the code of example alternation 4 in Section 15.3. How would you *specify* such a search so that termination is guaranteed even if key k does not occur in the database db? (In that case d may be given a value arbitrarily.)

Ex. 15.5 How would you implement the specification of Exercise 15.4?

Ex. 15.6 ♡ Use a type similar to that of example alternation 2 in Section 15.3 to modify your answer to Exercise 15.4 so that $\mathsf{ok}\, d$ is returned when $\mathsf{full}\, k\ \ d \in db$, and $\mathsf{undefined}$ is returned otherwise.

Ex. 15.7 How would you implement the specification Exercise 15.6?

Ex. 15.8 Write your answers to Exercises 15.5 and 15.7 as **loop** · **exit** · **end** constructions.

Ex. 15.9 ♡ Recall the type of Section 15.5

$$\textbf{type } \mathit{TooBig} \;\widehat{=}\; \mathsf{one} \mid \mathsf{several}\ (\textbf{set } \mathit{TooBig})\ ,$$

and consider the following subset of it:

$$\mathit{paradox} \;\widehat{=}\; \{t : \mathit{TooBig} \mid (\exists\, ts : \textbf{set } \mathit{TooBig} \cdot t = \mathsf{several}\, ts \wedge t \notin ts)\}\ .$$

Because *paradox* is a subset of *TooBig* we have that several *paradox* is an *element* of *TooBig*, and we can then reason

$$\begin{array}{ll}
 & \mathsf{several}\, \mathit{paradox} \in \mathit{paradox} \\
\equiv & \text{“definition of } \mathit{paradox} \text{ as set comprehension”} \\
 & (\exists\, ts : \textbf{set } \mathit{TooBig} \cdot \mathsf{several}\, \mathit{paradox} = \mathsf{several}\, ts \wedge \mathsf{several}\, \mathit{paradox} \notin ts) \\
\equiv & \text{“}\mathsf{several} \text{ is injective”} \\
 & (\exists\, ts : \textbf{set } \mathit{TooBig} \cdot \mathit{paradox} = ts \wedge \mathsf{several}\, \mathit{paradox} \notin ts) \\
\equiv & \text{“Predicate law ???”} \\
 & ???
\end{array}$$

What can we deduce at ??? What can we then conclude about *TooBig* itself?

(The above construction is an example of *Russell's paradox.*)

Ex. 15.10 Recall Exercise 15.9, concerning the type *TooBig*. Consider now the declaration

$$\textbf{type } \mathit{QuiteBig} \;\widehat{=}\; \mathsf{one} \mid \mathsf{several}\ (\textbf{finset } \mathit{QuiteBig})\ ,$$

which differs from *TooBig* only in its using **finset** in place of **set**. Replay from here the argument of Exercise 15.9; do we reach the same conclusion?

Ex. 15.11 In Section 15.6 we introduced the notion of (strict) partial order, together with its three defining properties (page 144). Show that, given *transitivity*, properties *irreflexivity* and *antisymmetry* are equivalent: each implies the other.

Ex. 15.12 A *non-strict* partial order is obtained from a strict one by allowing equality; thus we could define

$$x \circledleq y \;\widehat{=}\; x \circledless y \vee x = y\ . \tag{15.7}$$

Show that $\circledleq$, defined as above, satisfies these three properties:

- It is *reflexive*: for all x we have $x \circledleq x$;
- It is *antisymmetric*: for all x and y we have $x \circledleq y \wedge y \circledleq x \Rightarrow x = y$;
- It is *transitive*: for all x, y and z we have $x \circledleq y \wedge y \circledleq z \Rightarrow x \circledleq z$.

Then show that *any* non-strict partial order $\circledleq$ satisfying those three properties may be written as (15.7) for some strict partial order $\circledless$ satisfying the three properties of Section 15.6.

Ex. 15.13 In Exercise 15.12 there were three properties given to characterise a non-strict partial order. In Exercise 15.11 you were asked to show that the first two of the corresponding properties for a *strict* partial order were equivalent. Does that equivalence hold for the non-strict case as well? If not, propose a slight modification to the three (non-strict) properties so that equivalence of *reflexivity* and *antisymmetry* follows from *transitivity*.

Ex. 15.14 ♡ Is strict subset inclusion $\subset$ a strict partial order? Is it well founded?

Ex. 15.15 In Section 15.7 it was pointed out that *NatList* as defined at (15.3) is structurally equivalent to **seq** $\mathbb{N}$, and as a result we used from then on their notations interchangeably.

Can you define a type *Nat* that is structurally equivalent to $\mathbb{N}$ itself, and would justify the notation used in the following factorial calculator?

$$
\begin{array}{l}
f := 1; \\
\textbf{do } n \;\; \textbf{is } n' + 1 \rightarrow f, n := f \times n, n' \textbf{ od} \;.
\end{array}
$$

Ex. 15.16 ♡ Usually in Law *tagged iteration* 15.2 the variant is the expression E itself, and the relation $\otimes$ is 'is a component of'. Use the law to show that the following tagged iteration is a refinement of $r, s := \mathsf{rv}\, s, ?$, where rv is the list-reverse function:

$$
\begin{array}{l}
r := \langle\rangle; \\
\textbf{do } s \textbf{ is } h{:}t \rightarrow r, s := h{:}r, t \textbf{ od} \;.
\end{array}
$$

Hint: You will need a logical constant in the invariant.

Ex. 15.17 Consider the binary tree type

$$BT\,X \;\widehat{=}\; \mathsf{tip}\; X \mid \mathsf{node}\; (BT\,X)\; (BT\,X) \;.$$

The *frontier* of such a tree is the sequence of tip-values in left-to-right order. The following recursive procedure puts out the frontier of a given binary tree bt in $BT\;X$:

$$
\begin{array}{l}
\textbf{procedure } \mathit{Frontier}\; (\textbf{value } bt : BT\,X) \\
\widehat{=}\;\; \textbf{if } bt \textbf{ is} \\
\qquad\quad \mathsf{tip}\; a \rightarrow \textbf{output } a \\
\qquad [\!]\;\; \mathsf{node}\; bt1\;\; bt2 \rightarrow \\
\qquad\quad \mathit{Frontier}\;\; (bt1); \\
\qquad\quad \mathit{Frontier}\;\; (bt2) \\
\qquad \textbf{fi} \;.
\end{array}
$$

Develop *iterative* code (neither recursive nor parametrized) with the same effect, using a local variable $s : \textbf{seq}\, BT\,X$ as a stack. *Hint*: To express the invariant, use a recursive mathematical function $\mathsf{frontier}$ that you define yourself. You will need a logical constant.

Harder: What is the variant of your iteration?

Ex. 15.18 ♡ Recall Exercise 15.17. Develop *iterative* code, using two stacks, that determines whether the frontiers of two binary trees are equal:

$$\begin{array}{l} \textbf{var}\ bt1, bt2 : BT\ X; \\ \quad eq : Boolean; \end{array}$$

$$eq := (\mathsf{frontier}\ bt1 = \mathsf{frontier}\ bt2)$$

Your solution should be as space-efficient as possible: do not just 'output' into two sequences and then compare them.

Ex. 15.19 ♡ Verify the strict decrease of the variant $\mathsf{sizelt}\ ntl$ in Section 15.8.2.