latter, the proof objects either have to be expressed directly in the program or extracted as obligations and verified separately.

We now briefly reexamine the Curry-Howard isomorphism, when extended to the first-order level. We have the following correspondence:

Propositions	$\wedge$	$\supset$	$\top$	$\vee$	$\bot$	$\forall$	$\exists$
Types	$\times$	$\rightarrow$	$\mathbf{1}$	$+$	$\mathbf{0}$	Π	Σ

Note that under erasure, $\forall$ is related to $\rightarrow$ and $\exists$ is related to $\times$. The analogous property holds for Π and Σ: $\Pi x{:}\tau.\ \sigma$ corresponds to $\tau \rightarrow \sigma$ if x does not occur in σ, and $\Sigma x{:}\tau.\ \sigma$ simplifies to $\tau \times \sigma$ if x does not occur in σ.

In view of this strong correspondence, one wonders if propositions are really necessary as a primitive concept. In some systems, they are introduced in order to distinguish those elements with computational contents from those without. However, we have introduced the bracket annotation to serve this purpose, so one can streamline and simplify type theory by eliminating the distinction between propositions and types. Similarly, there is no need to distinguish between terms and proof terms. In fact, we have already used identical notations for them. Propositional constructs such as $n =_N m$ are then considered as types (namely: the types of their proof terms).

Because of the central importance of types and their properties in the design and theory of programming languages, there are many other constructions that are considered both in the literature and in practical languages. Just to name some of them, we have polymorphic types, singleton types, intersection types, union types, subtypes, record types, quotient types, equality types, inductive types, recursive types, linear types, strict types, modal types, temporal types, etc. Because of the essentially open-ended nature of type theory, all of these could be considered in the context of the machinery we have built up so far. We have seen most of the principles which underly the design of type systems (or corresponding logics), thereby providing a foundation for understanding the vast literature on the subject.

Instead of discussing these (which could be subject of another course) we consider one further application of dependent types and then consider theorem proving in various fragments of the full type theory.

4.9 Data Structure Invariants

An important application of dependent types is capturing representation invariants of data structures. An invariant on a data structure restricts valid elements of a type. Dependent types can capture such invariants, so that only valid elements are well-typed.

Our example will be an efficient implementation of finite sets of natural numbers. We start with a required lemma and auxiliary function.

$$\forall x{\in}\mathbf{nat}.\ \forall y{\in}\mathbf{nat}.\ [x < y] \vee [x =_N y] \vee [x > y]$$

From the straightforward proof we can extract a function

$$\mathit{compare} \in \mathbf{nat} \to \mathbf{nat} \to \mathbf{1} + \mathbf{1} + \mathbf{1}.$$

For obvious reasons we use the abbreviations

$$\begin{array}{rcl} \mathit{less} & = & \mathbf{inl}\,\langle\,\rangle \\ \mathit{equal} & = & \mathbf{inr}\,(\mathbf{inl}\,\langle\,\rangle) \\ \mathit{greater} & = & \mathbf{inr}\,(\mathbf{inr}\,\langle\,\rangle) \end{array}$$

and

$$\begin{array}{ll} \begin{array}{l} \mathbf{case}\ r \\ \quad \mathbf{of}\ \mathit{less} \Rightarrow t_< \\ \quad\ \ \mid \mathit{equal} \Rightarrow t_= \\ \quad\ \ \mid \mathit{greater} \Rightarrow t_> \\ \\ \end{array} & \begin{array}{l} = \mathbf{case}\ r \\ \quad \mathbf{of}\ \mathbf{inl}\,\_ \Rightarrow t_< \\ \quad\ \ \mid \mathbf{inr}\,r' \Rightarrow (\mathbf{case}\ r' \\ \qquad\qquad\qquad\quad \mathbf{of}\ \mathbf{inl}\,\_ \Rightarrow t_= \\ \qquad\qquad\qquad\quad\ \ \mid \mathbf{inr}\,\_ \Rightarrow t_>) \end{array} \end{array}$$

We give an interface for which we want to supply an implementation.

$$\begin{array}{rcl} \mathit{set} & & \mathit{type} \\ \mathit{empty} & \in & \mathit{set} \\ \mathit{insert} & \in & \mathbf{nat} \to \mathit{set} \to \mathit{set} \\ \mathit{member} & \in & \mathbf{nat} \to \mathit{set} \to \mathbf{bool} \end{array}$$

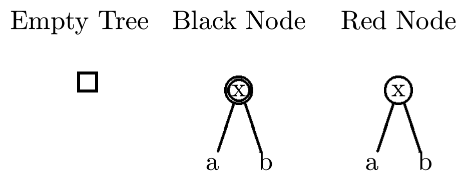
We do not give interfaces a first-class status in our development of type theory, but it is nonetheless a useful conceptual device. We would like to given an implementation via definitions of the form

$$\begin{array}{rcl} \mathit{set} & = & \ldots \\ \mathit{empty} & = & \ldots \\ \mathit{insert} & = & \ldots \\ \mathit{member} & = & \ldots \end{array}$$

that satisfy the types specified in the interface.

The idea is to implement a set as a *red-black tree*. Red-black trees are an efficient data structure for representing dictionaries whose keys are ordered. Here we follow the presentation by Chris Okasaki [Oka99]. The underlying data structure is a binary tree whose nodes are labelled with the members of the set. If we can ensure that the tree is sufficiently balanced, the height of such a tree will be logarithmic in the number of elements of the set. If we also maintain that the tree is ordered, lookup and insertion of an element can be performed in time proportional to the logarithm of the size of the set. The mechanism for ensuring that the tree remains balanced is the coloring of the nodes and the invariants maintained in the representation.

A tree is either empty or consists of a black or red node labelled with a natural number x and two subtrees a and b



We maintain the following representation invariants:

1. The tree is *ordered*: all elements in the left subtree a are smaller than x, while all elements in the right subtree b are larger than x.

2. The tree is *uniform*: every path from the root to a leaf contains the same number of black nodes. This defines the *black height* of a tree, where the black height of the empty tree is taken to be zero.

3. The tree is *well-colored*: the children of red node are always black, where empty trees count as black.

Uniform and well-colored trees are sufficiently balanced to ensure worst-case logarithmic membership test for elements in the set. Other operations can be implemented with similar efficiency, but we concentrate on membership test and insertion.

The ordering invariant is difficult to enforce by dependent types, since it requires propositional reasoning about the less-than relation. We will capture the uniformity invariant via dependent types. It is also possible to capture the coloring invariant via dependencies, but this is more complicated, and we do not attempt it here.

We index a red-black tree by its black height.

$$\frac{\Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash \mathit{tree}(n)\ \mathit{type}}\ \mathit{treeF}$$

There are three introduction rules, incorporating the three types of nodes (empty, black, red).

$$\frac{}{\Gamma \vdash \mathbf{E} \in \mathit{tree}(\mathbf{0})}\ \mathit{treeI}_E$$

$$\frac{\Gamma \vdash a \in \mathit{tree}(n) \qquad \Gamma \vdash x \in \mathbf{nat} \qquad \Gamma \vdash b \in \mathit{tree}(n)}{\Gamma \vdash \mathbf{B}\ a\ x\ b \in \mathit{tree}(\mathbf{s}(n))}\ \mathit{treeI}_B$$

$$\frac{\Gamma \vdash a \in \mathit{tree}(n) \qquad \Gamma \vdash x \in \mathbf{nat} \qquad \Gamma \vdash b \in \mathit{tree}(n)}{\Gamma \vdash \mathbf{R}\ a\ x\ b \in \mathit{tree}(n)}\ \mathit{treeI}_R$$

The index is increased by one for a black node $\mathbf{B}$, but not for a red node $\mathbf{R}$. Note that in either case, both subtrees a and b must have the same black height. This use of indices is different from their use for lists. Any list formed from **nil** and cons (::) without the length index will in fact have a valid length. Here, there are many trees that are ruled out as invalid because of the dependent types. In other words, the dependent types guarantee a data structure invariant by type-checking alone.

Now we can begin filling in the implementation, according to the given interface.

$$\begin{array}{rcl} \mathit{set} & = & \Sigma n{\in}\mathbf{nat}.\ \mathit{tree}(n) \\ \mathit{empty} & = & \langle \mathbf{0}, \mathbf{E} \rangle \\ \mathit{insert} & = & \ldots \\ \mathit{member} & = & \ldots \end{array}$$

Our intent is not to carry the black height n at run-time. If we wanted to make this explicit, we would write $\Sigma[n]{\in}[\mathbf{nat}].\ \mathit{tree}[n]$.

Before we program the *insert* and *member* functions, we write out the elimination form as a schema of primitive recursion.

$$\begin{array}{rcl} f(\mathbf{0}, \mathbf{E}) & = & t_E \\ f(\mathbf{s}(n'), \mathbf{B}\ a\ x\ b) & = & t_B(n', a, x, b, f(n', a), f(n', b)) \\ f(n, \mathbf{R}\ a\ x\ b) & = & t_R(n, a, x, b, f(n, a), f(n, b)) \end{array}$$

Using this schema, we can define the set membership function.

$$\mathit{mem} \ : \ \mathbf{nat} \rightarrow \Pi n{\in}\mathbf{nat}.\ \mathit{tree}(n) \rightarrow \mathbf{bool}$$

$$\begin{array}{llrrcl} \mathit{mem} & x & \mathbf{0} & \mathbf{E} & = & \mathit{false} \\ \mathit{mem} & x & (\mathbf{s}(n')) & (\mathbf{B}\ a\ y\ b) & = & \mathbf{case}\ \mathit{compare}\ x\ y \\ & & & & & \quad \mathbf{of}\ \mathit{less} \Rightarrow \mathit{mem}\ x\ n'\ a \\ & & & & & \qquad \mid \mathit{equal} \Rightarrow \mathit{true} \\ & & & & & \qquad \mid \mathit{greater} \Rightarrow \mathit{mem}\ x\ n'\ b \\ \mathit{mem} & x & n & (\mathbf{R}\ a\ y\ b) & = & \mathbf{case}\ \mathit{compare}\ x\ y \\ & & & & & \quad \mathbf{of}\ \mathit{less} \Rightarrow \mathit{mem}\ x\ n\ a \\ & & & & & \qquad \mid \mathit{equal} \Rightarrow \mathit{true} \\ & & & & & \qquad \mid \mathit{greater} \Rightarrow \mathit{mem}\ x\ n\ b \end{array}$$

Note that the cases for black and red nodes are identical, except for their treatment of the indices. This is the price we have to pay for our representation. However, in practice this can be avoided by allowing some type inference rather than just type checking.

From *mem* we can define the *member* function:

$$\begin{array}{rcl} \mathit{member} & \in & \mathbf{nat} \rightarrow \mathit{set} \rightarrow \mathbf{bool} \\ \mathit{member} & = & \lambda x{\in}\mathbf{nat}.\ \lambda s{\in}\mathit{set}.\ \mathbf{let}\ \langle n, t \rangle = s\ \mathbf{in}\ \mathit{mem}\ x\ n\ t \end{array}$$

Insertion is a much trickier operation. We have to temporarily violate our color invariant and then restore it by a re-balancing operation. Moreover, we sometimes need to increase the black height of the tree (essentially, when we run out of room at the current level). We need an auxiliary function

$$\mathit{ins} \ \in \ \mathbf{nat} \rightarrow \Pi n{\in}\mathbf{nat}.\ \mathit{tree}(n) \rightarrow \mathit{tree}(n)$$

which preserves the black height n, but may violate the red-black invariant at the root. That is, the resulting tree must be a valid red-black tree, except that the root might be red and either the left or the right subtree could also have a red root. At the top-level, we re-establish the color invariant by re-coloring the root black. We first show this step, assuming a function *ins* according to our specification above. Recall that $\mathit{set} = \Sigma n{\in}\mathbf{nat}.\ \mathit{tree}(n)$

$$\mathit{recolor} \in \Pi n{\in}\mathbf{nat}.\ \mathit{tree}(n) \rightarrow \mathit{set}$$

$$\begin{array}{lrrcl}
\mathit{recolor} & \mathbf{0} & \mathbf{E} & = & \langle \mathbf{0}, \mathbf{E} \rangle \\
\mathit{recolor} & (\mathbf{s}(n')) & (\mathbf{B}\ a\ x\ b) & = & \langle \mathbf{s}(n'), \mathbf{B}\ a\ x\ b \rangle \\
\mathit{recolor} & n & (\mathbf{R}\ a\ x\ b) & = & \langle \mathbf{s}(n), \mathbf{B}\ a\ x\ b \rangle
\end{array}$$

$$\begin{array}{lcl}
\mathit{insert} & \in & \mathbf{nat} \rightarrow \mathit{set} \rightarrow \mathit{set} \\
\mathit{insert} & = & \lambda x \in \mathbf{nat}.\ \lambda s \in \mathit{set}. \\
 & & \quad \mathbf{let}\ \langle n, t \rangle = s\ \mathbf{in}\ \mathit{recolor}\ n\ (\mathit{ins}\ x\ t)
\end{array}$$

Note that *recolor* returns a tree of black height n if the root node is black, and $\mathbf{s}(n)$ if the root node is red. This is how the black height can grow after successive insertion operations.

Now to the auxiliary function *ins*. Recall:

$$\mathit{ins} \in \mathbf{nat} \rightarrow \Pi n{\in}\mathbf{nat}.\ \mathit{tree}(n) \rightarrow \mathit{tree}(n)$$

It is critical that the black height of the output tree is the same as the input tree, so that the overall balance of the tree is not compromised during the recursion. This forces, for example, the case of insertion into an empty tree to color the new node red.

$$\begin{array}{llrrcl}
\mathit{ins} & x & \mathbf{0} & \mathbf{E} & = & \mathbf{R}\ \mathbf{E}\ x\ \mathbf{E} \\
\mathit{ins} & x & (\mathbf{s}(n')) & (\mathbf{B}\ a\ y\ b) & = & \mathbf{case}\ \mathit{compare}\ x\ y \\
 & & & & & \quad \mathbf{of}\ \mathit{less} \Rightarrow \mathit{balance}_L\ n'\ (\mathit{ins}\ x\ n'\ a)\ y\ b \\
 & & & & & \quad\ \ \mid \mathit{equal} \Rightarrow \mathbf{B}\ a\ y\ b \\
 & & & & & \quad\ \ \mid \mathit{greater} \Rightarrow \mathit{balance}_R\ n'\ a\ y\ (\mathit{ins}\ x\ n'\ a) \\
\mathit{ins} & x & n & (\mathbf{R}\ a\ y\ b) & = & \mathbf{case}\ \mathit{compare}\ x\ y \\
 & & & & & \quad \mathbf{of}\ \mathit{less} \Rightarrow \mathbf{R}\ (\mathit{ins}\ x\ n\ a)\ y\ b \\
 & & & & & \quad\ \ \mid \mathit{equal} \Rightarrow \mathbf{R}\ a\ y\ b \\
 & & & & & \quad\ \ \mid \mathit{greater} \Rightarrow \mathbf{R}\ a\ y\ (\mathit{ins}\ x\ n\ a)
\end{array}$$

We need two auxiliary functions $\mathit{balance}_L$ and $\mathit{balance}_R$ to repair any possible violation of the color invariant in either the left or right subtree, in case the root node is black.

$$\mathit{balance}_L \in \Pi n'{\in}\mathbf{nat}.\ \mathit{tree}(n') \to \mathbf{nat} \to \mathit{tree}(n') \to \mathit{tree}(\mathbf{s}(n'))$$

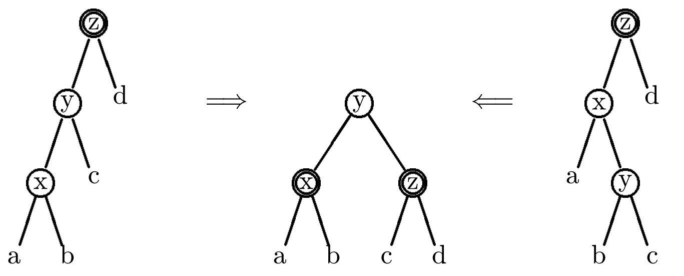
$$\begin{array}{rcll}
\mathit{balance}_L\ n'\ (\mathbf{R}\ (\mathbf{R}\ a\ x\ b)\ y\ c)\ z\ d & = & \mathbf{B}\ (\mathbf{R}\ a\ x\ b)\ y\ (\mathbf{R}\ c\ z\ d) & \\
\mathit{balance}_L\ n'\ (\mathbf{R}\ a\ x\ (\mathbf{R}\ b\ y\ c))\ z\ d & = & \mathbf{B}\ (\mathbf{R}\ a\ x\ b)\ y\ (\mathbf{R}\ c\ z\ d) & \\
\mathit{balance}_L\ n'\ a\ x\ b & = & \mathbf{B}\ a\ x\ b & \text{in all other cases}
\end{array}$$

$$\mathit{balance}_R \in \Pi n'{\in}\mathbf{nat}.\ \mathit{tree}(n') \to \mathbf{nat} \to \mathit{tree}(n') \to \mathit{tree}(\mathbf{s}(n'))$$

$$\begin{array}{rcll}
\mathit{balance}_R\ n'\ a\ x\ (\mathbf{R}\ (\mathbf{R}\ b\ y\ c)\ z\ d) & = & \mathbf{B}\ (\mathbf{R}\ a\ x\ b)\ y\ (\mathbf{R}\ c\ z\ d) & \\
\mathit{balance}_R\ n'\ a\ x\ (\mathbf{R}\ b\ y\ (\mathbf{R}\ c\ z\ d)) & = & \mathbf{B}\ (\mathbf{R}\ a\ x\ b)\ y\ (\mathbf{R}\ c\ z\ d) & \\
\mathit{balance}_R\ n'\ a\ x\ b & = & \mathbf{B}\ a\ x\ b & \text{in all other cases}
\end{array}$$

We have taken the liberty of combining some cases to significantly simplify the specification. It should be clear that this can indeed be implemented. In fact, there is no recursion, only several nested case distinctions.

The following picture illustrates the operation performed by $\mathit{balance}_L$. Note that the tree input trees to the left and the right are never actually built, but that $\mathit{balance}_L$ directly receives the left subtree, z and d as arguments.



Type-checking will verify that the black-height remains invariant under the balancing operation: initially, it is n' for each subtree a, b, c, and d and $\mathbf{s}(n')$ for the whole tree, which is still the case after re-balancing.

Similarly, the order is preserved. Writing $t < x$ to mean that every element in tree t is less than x, we extract the order

$$a < x < b < y < c < z < d$$

from all three trees by traversing it in a "smallest first" fashion.

Finally, we can see that the tree resulting from balancing always satisfies the red-black invariant, if the pictures on the left and right indicate the *only* place where the invariant is violated before we start.

All these proofs can be formalized, using an appropriate formalization of these color and ordering invariants. The only important consideration we have not mentioned is that in the case of insertion into a tree with a red root, we do not to apply the re-balancing operation to the result. This is because (a) the two immediate subtrees must be black, and (b) inserting into a black tree always yields a valid red-black tree (with *no* possible violation at the root).

This example illustrates how dependent types can be used to enforce a continuum of properties via type-checking, while others are left to explicit proof. From the software engineering point of view, any additional invariants that can be enforced statically without an explosion in the size of the program is likely to be beneficial by catching programming errors early.

Bibliography

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.