# Chapter 1

# Recursive Types

In programming in a practical functional language, there often arises a need for *recursive data structures* (or *inductive data structures*) whose components are data structures of the same kind but of smaller size. For example, a tree is a recursive data structure because children of the root node may be smaller trees of the same kind. In fact, we may even think of natural numbers as recursive data structures because a non-zero natural number can be expressed as a successor of another natural number.

The type system developed so far, however, cannot account for recursive data structures. Intuitively types for recursive data structures require recursive definitions at the level of types, but the previous type system does not provide such a language construct. (Recursive definitions at the level of expressions can be expressed by the fixed point construct.) This chapter introduces a new language construct for declaring *recursive types* which express recursive definitions at the level of types.

With recursive types, we can declare types for recursive data structures. For example, we declare a recursive type $\mathsf{ntree}$ for binary trees of natural numbers (of type $\mathsf{nat}$) with the following recursive definition:

$$\mathsf{ntree} \quad \cong \quad \mathsf{nat}+(\mathsf{ntree} \times \mathsf{ntree})$$

That is, $\mathsf{ntree}$ is either a single natural number of type $\mathsf{nat}$ (corresponding to leaf nodes) or two such binary trees of type $\mathsf{ntree}$ (corresponding to internal nodes).

There are two approaches to formalizing recursive types: *equi-recursive* and *iso-recursive* approaches which differ in the interpretation of $\cong$ in recursive definitions of types. Under the equi-recursive approach, $\cong$ stands for an equality relation. For example, the recursive definition of $\mathsf{ntree}$ specifies that $\mathsf{ntree}$ and $\mathsf{nat}+(\mathsf{ntree} \times \mathsf{ntree})$ are equal and thus interchangeable: $\mathsf{ntree}$ is automatically (*i.e.*, without the intervention of programmers) converted to $\mathsf{nat}+(\mathsf{ntree} \times \mathsf{ntree})$ and vice versa as necessary to make a given expression typecheck. Under the iso-recursive approach, $\cong$ stands for an isomorphic relation: two types in a recursive definition cannot be identified, but can be converted to each other by certain functions. For example, the recursive definition of $\mathsf{ntree}$ declares two functions for converting between $\mathsf{ntree}$ and $\mathsf{nat}+(\mathsf{ntree} \times \mathsf{ntree})$:

$$\begin{array}{rcl} \mathsf{fold}_{\mathsf{ntree}} & : & \mathsf{nat}+(\mathsf{ntree} \times \mathsf{ntree}) \rightarrow \mathsf{ntree} \\ \mathsf{unfold}_{\mathsf{ntree}} & : & \mathsf{ntree} \rightarrow \mathsf{nat}+(\mathsf{ntree} \times \mathsf{ntree}) \end{array}$$

To create a value of type $\mathsf{ntree}$, we first create a value of type $\mathsf{nat}+(\mathsf{ntree} \times \mathsf{ntree})$ (depending on whether it consists of a single leaf node or not), and then apply function $\mathsf{fold}_{\mathsf{ntree}}$; to analyze a value of type $\mathsf{ntree}$, we first apply function $\mathsf{unfold}_{\mathsf{ntree}}$ and then analyze the resultant value using a case expression.

Below we formalize recursive types under the iso-recursive approach. We will also see that SML uses the iso-recursive approach to deal with datatype declarations.

## 1.1 Definition

Consider the recursive definition of ntree. We may think of ntree as the solution to the following equation where $\alpha$ is a type variable standing for "any type" as in polymorphic type systems:

$$\alpha \quad \cong \quad \mathsf{nat}+(\alpha \times \alpha)$$

Since substituting ntree for $\alpha$ yields the original recursive definition of ntree, ntree is indeed the solution to the above equation. We choose to write $\mu\alpha.\mathsf{nat}+(\alpha \times \alpha)$ for the solution to the above equation where $\alpha$ is a fresh type variable. Then we can redefine ntree as follows:

$$\mathsf{ntree} \quad = \quad \mu\alpha.\mathsf{nat}+(\alpha \times \alpha)$$

Generalizing the example of ntree, we use a recursive type $\mu\alpha.A$ for the solution to the equation $\alpha \cong A$ where $A$ may contain occurrences of type variable $\alpha$:

$$\begin{array}{lrcl} \mathsf{type} & A & ::= & \cdots \mid \alpha \mid \mu\alpha.A \end{array}$$

The intuition is that $C = \mu\alpha.A$ means $C \cong [C/\alpha]A$. Since $\mu\alpha.A$ declares a fresh type variable $\alpha$ which is valid only within $A$, not every recursive type qualifies as a valid type. For example, $\mu\alpha.\alpha+\beta$ is not a valid recursive type unless it is part of another recursive type declaring type variable $\beta$. In order to be able to check the validity of a given recursive type, we define a typing context as an ordered set of type bindings and type declarations, as in polymorphic type systems:

$$\begin{array}{lrcl} \text{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \mathsf{type} \end{array}$$

The rules for the type judgment $\Gamma \vdash A$ type are similar to those for polymorphic types:

$$\frac{\alpha\ \mathsf{type} \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}}\ \mathsf{TyVar} \qquad \frac{\Gamma, \alpha\ \mathsf{type} \vdash A\ \mathsf{type}}{\Gamma \vdash \mu\alpha.A\ \mathsf{type}}\ \mathsf{Ty}\mu$$

Given a recursive type $C = \mu\alpha.A$, we need to be able to convert $[C/\alpha]A$ to $C$ and vice versa so as to create or analyze a value of type $C$. Thus, under the iso-recursive approach, a declaration of a recursive type $C = \mu\alpha.A$ implicitly introduces two primitive constructs $\mathsf{fold}_C$ and $\mathsf{unfold}_C$ specialized for type $C$. Operationally we may think of $\mathsf{fold}_C$ and $\mathsf{unfold}_C$ as behaving like functions of following types:

$$\begin{array}{rcl} \mathsf{fold}_C & : & [C/\alpha]A \to C \\ \mathsf{unfold}_C & : & C \to [C/\alpha]A \end{array}$$

Since $\mathsf{fold}_C$ and $\mathsf{unfold}_C$ are actually not functions but primitive constructs which always require an additional expression as an argument (*e.g.*, we cannot treat $\mathsf{fold}_C$ as a first-class object), the abstract syntax is extended as follows:

$$\begin{array}{rrcl} \text{expression} & e & ::= & \cdots \mid \mathsf{fold}_A\ e \mid \mathsf{unfold}_A\ e \\ \text{value} & v & ::= & \cdots \mid \mathsf{fold}_A\ v \end{array}$$

The typing rules for $\mathsf{fold}_C$ and $\mathsf{unfold}_C$ are derived from the operational interpretation of $\mathsf{fold}_C$ and $\mathsf{unfold}_C$ given above:

$$\frac{C = \mu\alpha.A \quad \Gamma \vdash e : [C/\alpha]A \quad \Gamma \vdash C\ \mathsf{type}}{\Gamma \vdash \mathsf{fold}_C\ e : C}\ \mathsf{Fold} \quad \frac{C = \mu\alpha.A \quad \Gamma \vdash e : C}{\Gamma \vdash \mathsf{unfold}_C\ e : [C/\alpha]A}\ \mathsf{Unfold}$$

The following reduction rules are based on the eager reduction strategy:

$$\frac{e \mapsto e'}{\mathsf{fold}_A\ e \mapsto \mathsf{fold}_A\ e'}\ \mathit{Fold}$$

$$\frac{e \mapsto e'}{\mathsf{unfold}_A\ e \mapsto \mathsf{unfold}_A\ e'}\ \mathit{Unfold} \quad \frac{}{\mathsf{unfold}_A\ \mathsf{fold}_A\ v \mapsto v}\ \mathit{Unfold}^2$$

**Exercise 1.1.** Propose reduction rules for the lazy reduction strategy.

## 1.2 Recursive data structures

This section presents a few examples of translating datatype declarations of SML to recursive types. The key idea is that each datatype declaration in SML implicitly introduces a recursive type and that each data constructor belonging to datatype $A$ implicitly uses $\mathsf{fold}_A$ and each pattern match for datatype $A$ implicitly uses $\mathsf{unfold}_A$.

Let us begin with a non-recursive datatype which does not have to be translated to a recursive type:

$$\texttt{datatype bool} \;=\; \texttt{True} \mid \texttt{False}$$

Since a value of type bool is either True *or* False, we translate bool to a *sum* type unit+unit so as to express the existence of two alternatives; the use of type unit indicates that data constructors True and False require no argument:

$$\begin{array}{rcl}
\texttt{bool} & = & \mathsf{unit}+\mathsf{unit} \\
\texttt{True} & = & \mathsf{inl}_{\mathsf{unit}}\ () \\
\texttt{False} & = & \mathsf{inr}_{\mathsf{unit}}\ () \\
\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & = & \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ _.\ e_1 \mid \mathsf{inr}\ _.\ e_2
\end{array}$$

Thus data constructors, which are separated by $|$ in a datatype declaration, become separated by $+$ when translated to a type in the simply typed $\lambda$-calculus.

Now consider a recursive datatype for natural numbers:

$$\texttt{datatype nat} \;=\; \texttt{Zero} \mid \texttt{Succ}\ \mathsf{of}\ \texttt{nat}$$

A recursive type for nat is the solution to the equation $\texttt{nat} \cong \mathsf{unit}+\texttt{nat}$ where the left unit corresponds to Zero and the right nat corresponds to Succ:

$$\texttt{nat} \;=\; \mu\alpha.\mathsf{unit}+\alpha$$

Then both data constructors Zero and Succ first prepare a value of type unit+nat and then "fold" it to create a value of type nat:

$$\begin{array}{rcl}
\texttt{Zero} & = & \mathsf{fold}_{\texttt{nat}}\ \mathsf{inl}_{\texttt{nat}}\ () \\
\texttt{Succ}\ e & = & \mathsf{fold}_{\texttt{nat}}\ \mathsf{inr}_{\mathsf{unit}}\ e
\end{array}$$

A pattern match for datatype nat works in the opposite way: it first "unfolds" a value of type nat to obtain a value of type unit+nat and then uses a case expression to analyze it:

$$\mathsf{case}\ e\ \mathsf{of}\ \texttt{Zero} \Rightarrow e_1 \mid \texttt{Succ}\ x \Rightarrow e_2 \;=\; \mathsf{case}\ \mathsf{unfold}_{\texttt{nat}}\ e\ \mathsf{of}\ \mathsf{inl}\ \_.\ e_1 \mid \mathsf{inr}\ x.\ e_2$$

Similarly a recursive datatype for lists of natural numbers is translated as follows:

$$\begin{array}{rcl}
\texttt{datatype nlist} & = & \texttt{Nil} \mid \texttt{Cons}\ \mathsf{of}\ \texttt{nat} \times \texttt{nlist} \\
\texttt{nlist} & = & \mu\alpha.\mathsf{unit}+(\texttt{nat} \times \alpha) \\
\texttt{Nil} & = & \mathsf{fold}_{\texttt{nlist}}\ \mathsf{inl}_{\texttt{nat}\times\texttt{nlist}}\ () \\
\texttt{Cons}\ e & = & \mathsf{fold}_{\texttt{nlist}}\ \mathsf{inr}_{\mathsf{unit}}\ e \\
\mathsf{case}\ e\ \mathsf{of}\ \texttt{Nil} \Rightarrow e_1 \mid \texttt{Cons}\ x \Rightarrow e_2 & = & \mathsf{case}\ \mathsf{unfold}_{\texttt{nlist}}\ e\ \mathsf{of}\ \mathsf{inl}\ _.\ e_1 \mid \mathsf{inr}\ x.\ e_2
\end{array}$$

As an example of a recursive type that does not use a sum type, let us consider a datatype for streams of natural numbers:

$$\begin{array}{rcl}
\texttt{datatype nstream} & = & \texttt{Nstream}\ \mathsf{of}\ \mathsf{unit}\rightarrow\texttt{nat} \times \texttt{nstream} \\
\texttt{nstream} & = & \mu\alpha.\mathsf{unit}\rightarrow\texttt{nat} \times \alpha
\end{array}$$

When "unfolded," a value of type nstream yields a function of type $\mathsf{unit}\rightarrow\texttt{nat} \times \texttt{nstream}$ which returns a natural number and another stream. For example, the following $\lambda$-abstraction has type $\texttt{nstream}\rightarrow\texttt{nat} \times \texttt{nstream}$:

$$\lambda s\!:\!\texttt{nstream}.\ \mathsf{unfold}_{\texttt{nstream}}\ s\ ()$$

The following function, of type $\texttt{nat}\rightarrow\texttt{nstream}$, returns a stream of natural numbers beginning with its argument:

$$\lambda n\!:\!\texttt{nat}.\ (\mathsf{fix}\ f\!:\!\texttt{nat}\rightarrow\texttt{nstream}.\ \lambda x\!:\!\texttt{nat}.\ \mathsf{fold}_{\texttt{nstream}}\ \lambda y\!:\!\mathsf{unit}.\ (x, f\ (\texttt{Succ}\ x)))\ n$$

## 1.3 Typing the untyped $\lambda$-calculus

A further application of recursive types is a translation of the untyped $\lambda$-calculus to the simply typed $\lambda$-calculus augmented with recursive types. Specifically we wish to translate the untyped $\lambda$-calculus to the simply typed $\lambda$-calculus with the following definition:

$$\begin{array}{lrcl} \textsf{type} & A & ::= & A \rightarrow A \mid \alpha \mid \mu\alpha.A \\ \textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\; e \mid \textsf{fold}_A\; e \mid \textsf{unfold}_A\; e \end{array}$$

Note that unlike the pure simply typed $\lambda$-calculus, the definition of types does not include base types.

We translate an expression $e$ in the untyped $\lambda$-calculus to another expression $e^\circ$ in the simply typed $\lambda$-calculus. We treat all expressions in the untyped $\lambda$-calculus alike and assign a unique type $\Omega$ to every expression $e$ in the untyped $\lambda$-calculus (*i.e.*, $e^\circ$ is to have type $\Omega$). Then the key to the translation is to find such a unique type $\Omega$.

It is not difficult to find such a type $\Omega$ when recursive types are available. If every expression is assigned type $\Omega$, we may think that $\lambda x.\, e$ is assigned type $\Omega \rightarrow \Omega$ *as well as* type $\Omega$. Or, in order for $e_1\; e_2$ to be assigned type $\Omega$, $e_1$ must be assigned *not only* type $\Omega$ but also type $\Omega \rightarrow \Omega$ because $e_2$ is assigned type $\Omega$. Thus $\Omega$ must be identified with $\Omega \rightarrow \Omega$ (*i.e.*, $\Omega \cong \Omega \rightarrow \Omega$), and is defined as follows:

$$\Omega \quad = \quad \mu\alpha.\alpha \rightarrow \alpha$$

Then expressions in the untyped $\lambda$-calculus are translated as follows:

$$\begin{array}{rcl} x^\circ & = & x \\ (\lambda x.\, e)^\circ & = & \textsf{fold}_\Omega\; \lambda x\!:\!\Omega.\, e^\circ \\ (e_1\; e_2)^\circ & = & (\textsf{unfold}_\Omega\; {e_1}^\circ)\; {e_2}^\circ \end{array}$$

**Proposition 1.2.** $\cdot \vdash e^\circ : \Omega$ *holds for any expression $e$ in the untyped $\lambda$-calculus.*

**Proposition 1.3.** *If $e \mapsto e'$, then $e^\circ \mapsto^* e'^\circ$.*

In Proposition 1.3, extra reduction steps in $e^\circ \mapsto^* e'^\circ$ are due to applications of the rule $\mathit{Unfold}^2$.

An interesting consequence of the translation is that despite the absence of the fixed point construct, the reduction of an expression in the simply typed $\lambda$-calculus with recursive types may not terminate! For example, the reduction of $((\lambda x.\, x\; x)\; (\lambda x.\, x\; x))^\circ$ does not terminate because $(\lambda x.\, x\; x)\; (\lambda x.\, x\; x)$ reduces to itself. In fact, we can even write recursive functions — all we have to do is to translate the fixed point combinator fix (see Section **??**)!

## 1.4 Exercises

**Exercise 1.4.** Consider the simply typed $\lambda$-calculus augmented with recursive types. We use a function type $A \rightarrow B$ for non-recursive functions from type $A$ to type $B$. Now let us introduce another function type $A \Rightarrow B$ for *recursive* functions from type $A$ to type $B$. Define $A \Rightarrow B$ in terms of ordinary function types and recursive types.