Additional Exercises for Practical Foundations for Programming Languages (Second Edition)*

Robert Harper
Carnegie Mellon University

March 21, 2024

1 System T

The use of the recursor, rather than the iterator, as the elimination form for the type `nat` is necessary, for practical reasons, but unfortunate because it is essentially an *ad hoc* use of product types.

Exercises

1. Give the statics and dynamics of the iteration construct for **T**.
2. Define the predecessor operation on natural numbers using only iteration. (The predecessor of $\mathtt{z}$ is $\mathtt{z}$, and of $\mathtt{s}(\,e\,)$ is e.)

Solutions

1.

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma\, x : \tau \vdash e : \tau}{\Gamma \vdash \mathtt{iter}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(\,x\,) \hookrightarrow e_1\} : \tau} \tag{1}$$

$$\frac{}{\mathtt{iter}\, \mathtt{z}\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(\,x\,) \hookrightarrow e_1\} \longmapsto e_0} \tag{2}$$

$$\frac{}{\mathtt{iter}\, \mathtt{s}(\,e\,)\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(\,x\,) \hookrightarrow e_1\} \longmapsto \{\mathtt{iter}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(\,x\,) \hookrightarrow e_1\}/x\}e_1} \tag{3}$$

$$\frac{e \longmapsto e'}{\texttt{iter}\, e\, \{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(\, x\,) \hookrightarrow e_1\} \longmapsto \texttt{iter}\, e'\, \{\texttt{z} \hookrightarrow e_0 \mid \texttt{s}(\, x\,) \hookrightarrow e_1\}} \quad (4)$$

2. It is not directly definable using only the iterator, because it does not provide access to the predecessor. The simplest solution seems to be to define a pairing function on the natural numbers with which one can simulate the solution using products. (Whether it is possible to define pairing using only iteration needs to be checked; it involves defining basic arithmetic operations that plausibly need only iteration.)

2 Sums and Products

2.1 Algebraic Structure

There is a rich algebraic structure given by type isomorphisms involving sum and product types. At this early stage it is not possible to prove these results rigorously, but informally one may show that sum and product types each form a commutative monoid (up to isomorphism), and that the product distributes over sum. Using suggestive notations such as n for the n-fold sum of the type 1 with itself, one can show that the rules of grade school algebra hold as isomorphisms of types.

Type constructors have a functorial action on maps, which is explored in the following questions.

Exercises

1. For $f_1 : \sigma_1 \to \tau_1$ and $f_2 : \sigma_2 \to \tau_2$, define the following functions:
   (a) $\texttt{unit} : \texttt{unit} \to \texttt{unit}$.
   (b) $f_1 \times f_2 : (\, \sigma_1 \times \sigma_2\,) \to (\, \tau_1 \times \tau_2\,)$.
2. For $f_1 : \sigma_1 \to \tau_1$ and $f_2 : \sigma_2 \to \tau_2$, define the following functions:
   (a) $\texttt{void} : \texttt{void} \to \texttt{void}$.
   (b) $f_1 + f_2 : (\, \sigma_1 + \sigma_2\,) \to (\, \tau_1 + \tau_2\,)$.

Solutions

1. (a) $\texttt{unit} \triangleq \lambda\, (\, \_ : \texttt{unit}\,)\, \langle\rangle$.
   (b) $f_1 \times f_2 \triangleq \lambda\, (\, x : \sigma_1 \times \sigma_2\,)\, \langle f_1(\, x \cdot \texttt{l}\,), f_2(\, x \cdot \texttt{r}\,)\rangle$.
2. (a) $\texttt{void} \triangleq \lambda\, (\, x : \texttt{void}\,)\, \texttt{case}\, x\, \{\, \}$.
   (b) $f_1 + f_2 \triangleq \lambda\, (\, x : \sigma_1 + \sigma_2\,)\, \texttt{case}\, x\, \{\texttt{l} \cdot x_1 \hookrightarrow \texttt{l} \cdot f_1(\, x_1\,) \mid \texttt{r} \cdot x_2 \hookrightarrow \texttt{r} \cdot f_2(\, x_2\,)\}$.

Define $\sigma \cong \tau$ to mean there exists $f : \sigma \to \tau$ and $g : \tau \to \sigma$ that are, informally, mutually inverses. Argue that the following congruences are valid in the extension of **T** with nullary and binary products and sums.

Exercises

1. (a) $\sigma \cong \sigma$.
   (b) If $\sigma \cong \tau$, then $\tau \cong \sigma$.
   (c) If $\rho \cong \sigma$ and $\sigma \cong \tau$, then $\rho \cong \tau$.
   (d) $\sigma \times \tau \cong \tau \times \sigma$.
   (e) $\rho \times (\sigma \times \tau) \cong (\rho \times \sigma) \times \tau$.
   (f) $\sigma + \tau \cong \tau + \sigma$.
   (g) $\rho + (\sigma + \tau) \cong (\rho + \sigma) + \tau$.
   (h) $\mathtt{unit} \times \tau \cong \tau$.
   (i) $\mathtt{void} + \tau \cong \tau$.
   (j) $\rho \times (\sigma + \tau) \cong (\rho \times \sigma) + (\rho \times \tau)$.

2. (a) If $\sigma_1 \cong \tau_1$ and $\sigma_1 \cong \tau_2$, then $\sigma_1 \times \sigma_2 \cong \tau_1 \times \tau_2$.
   (b) If $\sigma_1 \cong \tau_1$ and $\sigma_1 \cong \tau_2$, then $\sigma_1 + \sigma_2 \cong \tau_1 + \tau_2$.
   (c) If $\sigma_1 \cong \tau_1$ and $\sigma_1 \cong \tau_2$, then $\sigma_1 \to \sigma_2 \cong \tau_1 \to \tau_2$.

3. (a) $\sigma \to (\tau_1 \times \tau_2) \cong (\sigma \to \tau_1) \times (\sigma \to \tau_2)$.
   (b) $(\sigma_1 + \sigma_2) \to \tau \cong (\sigma_1 \to \tau) \times (\sigma_2 \to \tau)$.
   (c) $\sigma \to \mathtt{unit} \cong \mathtt{unit}$.
   (d) $\sigma \to \mathtt{void} \cong \mathtt{void}$, provided that σ is inhabited.
   (e) $\mathtt{void} \to \tau \cong \mathtt{unit}$.
   (f) $\mathtt{unit} \to \tau \cong \tau$.

4. (a) Define 2 to be $\mathtt{unit} + \mathtt{unit}$. Show that $2 \times \tau \cong \tau + \tau$. More generally, defining n to be the n-fold sum of $\mathtt{unit}$, show by induction on n that $n \times \tau \cong \underbrace{\tau + \ldots + \tau}_{n}$.
   (b) Define τ^2 to be $\tau \times \tau$. Show that $\tau^2 \cong 2 \to \tau$. More generally, show that $\tau^n \cong n \to \tau$, where n is the n-fold sum of $\mathtt{unit}$ as defined above. In particular, $\tau^{\mathtt{void}} \cong \mathtt{unit}$.
   (c) Show that $(\tau + \mathtt{unit})^2 \cong \tau^2 + 2 \times \tau + \mathtt{unit}$. In other words, $\tau\ \mathtt{opt} \cong \sqrt{\tau^2 + 2 \times \tau + \mathtt{unit}}$.

Solutions

1. (a) Form the pair $\langle \lambda\,(\,x:\sigma\,)\,x, \lambda\,(\,x:\sigma\,)\,x\rangle$.
   (b) Given $\langle f, g\rangle : \sigma \cong \tau$, form $\langle g, f\rangle : \tau \cong \sigma$.
   (c) Given $\langle f, g\rangle : \rho \cong \sigma$ and $\langle h, i\rangle : \sigma \cong \rho$, form $\langle h \circ f, i \circ g\rangle : \rho \cong \tau$.
   (d) Form the pair $\langle \lambda\,(\,\langle x, y\rangle\,.\,\langle y, x\rangle\,), \lambda\,(\,\langle y, x\rangle\,.\,\langle x, y\rangle\,)\rangle : \sigma \times \tau \cong \tau \times \sigma$.
   (e) Form the pair $\langle \lambda\,(\,\langle x, \langle y, z\rangle\rangle\,.\,\langle\langle x, y\rangle, z\rangle\,), \lambda\,(\,\langle\langle x, y\rangle, z\rangle\,.\,\langle x, \langle y, z\rangle\rangle\,)\rangle$.
   (f) Form the pair $\langle f, g\rangle$, where
      i. $f \triangleq \lambda\,(\,x:\sigma+\tau\,)\,\mathtt{case}\,x\,\{\mathtt{l}\cdot x_1 \hookrightarrow \mathtt{r}\cdot x_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow \mathtt{l}\cdot x_2\}$, and
      ii. $g \triangleq \lambda\,(\,x:\tau+\sigma\,)\,\mathtt{case}\,y\,\{\mathtt{l}\cdot y_1 \hookrightarrow \mathtt{r}\cdot y_1 \mid \mathtt{r}\cdot y_2 \hookrightarrow \mathtt{l}\cdot y_2\}$.
   (g) Perform a nested case analysis, and reconstruct using the re-organized injections.
   (h) Form the pair $\langle \lambda\,(\,\langle\langle\rangle, x\rangle\,.\,x\,), \lambda\,(\,x:\tau\,)\,\langle\langle\rangle, x\rangle\rangle$.
   (i) Form the pair

   $$\langle \lambda\,(\,\mathtt{void}+\tau : x\,)\,\mathtt{case}\,x\,\{\mathtt{l}\cdot x_1 \hookrightarrow \mathtt{case}\,x_1\,\{\,\} \mid \mathtt{r}\cdot x_2 \hookrightarrow x_2\}, \lambda\,(\,x:\tau\,)\,\mathtt{r}\cdot x\rangle.$$

   (j) Form the pair $\langle f, g\rangle$, where
      i. $f \triangleq \lambda\,(\,\langle x, y\rangle\,.\,\mathtt{case}\,y\,\{\mathtt{l}\cdot y_1 \hookrightarrow \mathtt{l}\cdot\langle x, y_1\rangle \mid \mathtt{r}\cdot y_2 \hookrightarrow \mathtt{r}\cdot\langle x, y_2\rangle\}\,)$, and
      ii. $g \triangleq \lambda\,(\,z:(\,\rho\times\sigma\,)+(\,\rho\times\tau\,)\,)\,\mathtt{case}\,z\,\{\mathtt{l}\cdot\langle x, y\rangle \hookrightarrow \langle x, \mathtt{l}\cdot y\rangle \mid \mathtt{r}\cdot\langle x, z\rangle \hookrightarrow \langle x, \mathtt{r}\cdot z\rangle\}$.
2. (a) Given $\langle f_1, g_1\rangle : \sigma_1 \cong \tau_1$ and $\langle f_2, g_2\rangle : \sigma_2 \cong \tau_2$, form $\langle f_1 \times f_2, g_1 \times g_2\rangle$, and check that it is an isomorphism pair.
   (b) Given $\langle f_1, g_1\rangle : \sigma_1 \cong \tau_1$ and $\langle f_2, g_2\rangle : \sigma_2 \cong \tau_2$, form $\langle f_1 + f_2, g_1 + g_2\rangle$, and check that it is an isomorphism pair.
   (c) Given $\langle f_1, g_1\rangle : \sigma_1 \cong \tau_1$ and $\langle f_2, g_2\rangle : \sigma_2 \cong \tau_2$, form $\langle f_1 \to f_2, g_1 \to g_2\rangle$, and check that it is an isomorphism pair.
3. (a) Form the pair $\langle f, g\rangle$, where
      i. $f \triangleq \lambda\,(\,x:\sigma\to(\,\tau_1\times\tau_2\,)\,)\,\langle\lambda\,(\,x:\sigma\,)\,x\cdot\mathtt{l}, \lambda\,(\,x:\sigma\,)\,x\cdot\mathtt{r}\rangle$.
      ii. $g \triangleq \lambda\,(\,\langle z_1, z_2\rangle\,.\,\lambda\,(\,x:\sigma\,)\,\langle z_1(\,x\,), z_2(\,x\,)\rangle\,)$.
   (b) Form the pair $\langle f, g\rangle$, where
      i. $f \triangleq \lambda\,(\,x:(\,\sigma_1+\sigma_2\,)\to\tau\,)\,\langle\lambda\,(\,x_1:\sigma_1\,)\,x(\,\mathtt{l}\cdot x_1\,), \lambda\,(\,x_2:\sigma_2\,)\,x(\,\mathtt{r}\cdot x_2\,)\rangle$.
      ii. $g \triangleq \lambda\,(\,\langle z_1, z_2\rangle\,.\,\lambda\,(\,x:\sigma_1+\sigma_2\,)\,\mathtt{case}\,x\,\{\mathtt{l}\cdot x_1 \hookrightarrow z\cdot\mathtt{l}(\,x_1\,) \mid \mathtt{r}\cdot x_2 \hookrightarrow z\cdot\mathtt{r}(\,x_2\,)\}\,)$.
   (c) Form the pair $\langle \lambda\,(\,z:\sigma\to\mathtt{unit}\,)\,\langle\rangle, \lambda\,(\,\_:\mathtt{unit}\,)\,\lambda\,(\,\_:\sigma\,)\,\langle\rangle\rangle$.
   (d) Form the pair $\langle \lambda\,(\,z:\sigma\to\mathtt{void}\,)\,z(\,e_\sigma\,), \lambda\,(\,z:\mathtt{void}\,)\,\mathtt{case}\,z\,\{\,\}\rangle$, where $e_\sigma : \sigma$.
   (e) Form the pair $\langle \lambda\,(\,z:\mathtt{void}\to\tau\,)\,\langle\rangle, \lambda\,(\,z:\mathtt{unit}\,)\,\lambda\,(\,x:\mathtt{void}\,)\,\mathtt{case}\,x\,\{\,\}\rangle$.

(f) Form the pair $\langle \lambda\,(\,z : \mathtt{unit} \to \tau\,)\, z(\,\langle\rangle\,), \lambda\,(\,x : \mathtt{unit}\,)\, \lambda\,(\,y : \mathtt{unit}\,)\, x \rangle$.

4. (a) Using the foregoing isomorphisms, show that $2 \times \tau \cong \tau + \tau$:

$$\begin{aligned} 2 \times \tau &\cong (\,\mathtt{unit} + \mathtt{unit}\,) \times \tau \\ &\cong (\,\mathtt{unit} \times \tau\,) + (\,\mathtt{unit} \times \tau\,) \\ &\cong \tau + \tau. \end{aligned}$$

(b) Using the foregoing isomorphisms, show that $\tau^2 \cong 2 \to \tau$:

$$\begin{aligned} 2 \to \tau &\cong (\,\mathtt{unit} + \mathtt{unit}\,) \to \tau \\ &\cong (\,\mathtt{unit} \to \tau\,) \times (\,\mathtt{unit} \to \tau\,) \\ &\cong \tau \times \tau \\ &= \tau^2. \end{aligned}$$

(c) Using the foregoing isomorphisms, show that $\tau\ \mathtt{opt} \cong \sqrt{\tau^2 + 2 \times \tau + \mathtt{unit}}$:

$$\begin{aligned} (\,\tau\ \mathtt{opt}\,)^2 &= (\,\tau + \mathtt{unit}\,)^2 \\ &\cong (\,\tau + \mathtt{unit}\,) \times (\,\tau + \mathtt{unit}\,) \\ &\cong (\,(\,\tau + \mathtt{unit}\,) \times \tau\,) + (\,(\,\tau + \mathtt{unit}\,) \times \mathtt{unit}\,) \\ &\cong (\,(\,\tau \times \tau\,) + (\,\mathtt{unit} \times \tau\,)\,) + (\,(\,\mathtt{unit} \times \tau\,) + (\,\mathtt{unit} \times \mathtt{unit}\,)\,) \\ &\cong \tau^2 + (\,\tau + \tau\,) + \mathtt{unit} \\ &\cong \tau^2 + 2 \times \tau + \mathtt{unit} \end{aligned}$$

Classification

Sums are used to classify the values of a type. For example, a complex number may be classified as either cartesian or polar, according to whether it is given by its (x, y)-coordinates or its (ρ, θ)-coordinates. These two representations should not be confused; the cartesian and the polar are two *classes* of the type of complex numbers. Classes should not be confused with types, as is invariably done in object-oriented languages.

First, let us define the types of complex numbers and of the operations on them.

$$\begin{aligned} \tau^{\mathtt{cplx}} &= [\mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}}] \\ \tau^{\mathtt{cart}} &= \langle \mathtt{x} \hookrightarrow \mathtt{float}, \mathtt{y} \hookrightarrow \mathtt{float} \rangle \\ \tau^{\mathtt{pol}} &= \langle \mathtt{r} \hookrightarrow \mathtt{float}, \mathtt{t} \hookrightarrow \mathtt{float} \rangle \\ \rho_{\mathtt{opns}} &= \langle \mathtt{quad} \hookrightarrow \rho_{\mathtt{quad}}, \mathtt{dist} \hookrightarrow \rho_{\mathtt{dist}} \rangle \\ \rho_{\mathtt{quad}} &= [\mathtt{I}, \mathtt{II}, \mathtt{III}, \mathtt{IV}] \\ \rho_{\mathtt{dist}} &= \mathtt{float} \end{aligned}$$

Thus, a complex number is either cartesian or polar, from which its squared distance from the origin or the quadrant may be determined.

Let us analyze the essential structure of functions of type $\tau^{\mathtt{cplx}} \to \rho_{\mathtt{opns}}$, which compute the operations on a complex number. The plan is to mimic the situation in linear algebra in which a linear transformation between vector spaces can be represented as a matrix. The algebra of products and sums allows us to formulate the foregoing function type as a matrix in two equivalent ways, isolating the essential content of such a mapping.

Because the domain is a sum and the range is a product, the function type may be decomposed into a matrix in either row-major or column-major order.

Exercises

1. Give the class-first decomposition of the type $\tau^{\mathtt{cplx}} \to \rho_{\mathtt{opns}}$ as a product type:
$$\tau^{\mathtt{cplx}} \to \rho_{\mathtt{opns}} \cong \ldots$$
2. Give the operations-first decomposition of the type $\tau^{\mathtt{cplx}} \to \rho_{\mathtt{opns}}$ as a product type:
$$\tau^{\mathtt{cplx}} \to \rho_{\mathtt{opns}} \cong \ldots$$

Solutions

1. Give the class-first decomposition of the type $\tau^{\mathtt{cplx}} \to \rho_{\mathtt{opns}}$ as a product type:
$$\tau^{\mathtt{cplx}} \to \rho_{\mathtt{opns}} \cong \langle \mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}} \to \rho_{\mathtt{opns}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}} \to \rho_{\mathtt{opns}} \rangle$$
2. Give the operations-first decomposition of the type $\tau^{\mathtt{cplx}} \to \rho_{\mathtt{opns}}$ as a product type:
$$\tau^{\mathtt{cplx}} \to \rho_{\mathtt{opns}} \cong \langle \mathtt{quad} \hookrightarrow \tau^{\mathtt{cplx}} \to \rho_{\mathtt{quad}}, \mathtt{dist} \hookrightarrow \tau^{\mathtt{cplx}} \to \rho_{\mathtt{dist}} \rangle$$

These decompositions involve four different function types to which the same decompositions may be applied to obtain product types:

Exercises

1. Give the product decomposition of the class-first component types:
$$\tau^{\mathtt{cart}} \to \rho_{\mathtt{opns}} \cong \langle \mathtt{quad} \hookrightarrow \tau^{\mathtt{cart}} \to \rho_{\mathtt{quad}}, \mathtt{dist} \hookrightarrow \tau^{\mathtt{cart}} \to \rho_{\mathtt{dist}} \rangle$$
$$\tau^{\mathtt{pol}} \to \rho_{\mathtt{opns}} \cong \langle \mathtt{quad} \hookrightarrow \tau^{\mathtt{pol}} \to \rho_{\mathtt{quad}}, \mathtt{dist} \hookrightarrow \tau^{\mathtt{pol}} \to \rho_{\mathtt{dist}} \rangle$$

2. Give the product decomposition of the operation-first component types:

$$\tau^{\mathtt{cplx}} \to \rho_{\mathtt{quad}} \cong \langle \mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}} \to \rho_{\mathtt{quad}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}} \to \rho_{\mathtt{quad}} \rangle$$
$$\tau^{\mathtt{cplx}} \to \rho_{\mathtt{dist}} \cong \langle \mathtt{cart} \hookrightarrow \tau^{\mathtt{cart}} \to \rho_{\mathtt{dist}}, \mathtt{pol} \hookrightarrow \tau^{\mathtt{pol}} \to \rho_{\mathtt{dist}} \rangle$$

Combining these decompositions two different two-by-two matrix representations of the function type $\tau^{\mathtt{cplx}} \to \rho^{\mathtt{opns}}$ may be obtained. Neither organization is preferable or superior to the other; in fact, they are both isomorphic to the function type.

3 Generic Programming

The addition of variable types allows the formulation of *type polynomials* that obey many of the rules of algebra in the sense of type isomorphism. Type polynomials are central to the concept of *generic*, or *type-directed*, programming, which is induced by the functorial action of type constructors on mappings.

The generic extension of a map along a polynomial type operator can be neatly expressed using the action of type constructors on maps. It is helpful to use the "curried" form of generic extension, $\mathtt{map}[\, t \,.\, \tau \,](\, x \,.\, e \,)$, of type $\{\rho/t\}\tau \to \{\sigma/t\}\tau$, where $x : \rho \vdash e : \sigma$.

Exercises

1. Define the generic extension along product types:

$$\mathtt{map}[\, t \,.\, \tau_1 \times \tau_2 \,](\, x \,.\, e \,) \longmapsto \ldots$$
$$\mathtt{map}[\, t \,.\, \mathtt{unit} \,](\, x \,.\, e \,) \longmapsto \ldots$$

2. Define the generic extension along sum types:

$$\mathtt{map}[\, t \,.\, \tau_1 + \tau_2 \,](\, x \,.\, e \,) \longmapsto \ldots$$
$$\mathtt{map}[\, t \,.\, \mathtt{void} \,](\, x \,.\, e \,) \longmapsto \ldots$$

Solutions

1. Define the generic extension along product types:

$$\mathtt{map}[\, t \,.\, \tau_1 \times \tau_2 \,](\, x \,.\, e \,) \longmapsto \mathtt{map}[\, t \,.\, \tau_1 \,](\, x \,.\, e \,) \times \mathtt{map}[\, t \,.\, \tau \,](\, x \,.\, e \,)$$
$$\mathtt{map}[\, t \,.\, \mathtt{unit} \,](\, x \,.\, e \,) \longmapsto \mathtt{unit}$$

2. Define the generic extension along sum types:

$$\texttt{map}[\,t\,.\,\tau_1 + \tau_2\,](\,x\,.\,e\,) \longmapsto \texttt{map}[\,t\,.\,\tau_1\,](\,x\,.\,e\,) + \texttt{map}[\,t\,.\,\tau_2\,](\,x\,.\,e\,)$$
$$\texttt{map}[\,t\,.\,\texttt{void}\,](\,x\,.\,e\,) \longmapsto \texttt{void}$$

4 Inductive and Coinductive Types

Generic programming is key to giving a general account of inductive (finitary) and coinductive (infinitary) types. Traditional "box and pointer" diagrams make sense only for a limited case of inductive types (those with eager constructors), and are completely useless for dealing with laziness, coinductive types, or functions. It is important to expose students to the far richer world of types and their programs considered here.

The definitions of inductive and coinductive types makes use of generic extension. Having defined these, it is then a interesting exercise to extend generic extension to account for inductive and coinductive types. The interesting part of the problem is to ensure that the generic extension is well-defined. In the case of a polynomial type operator this is ensured by induction on the structure of types. But when extending to inductive and coinductive types the justification is not so straightforward, essentially because it must appeal to the generic extension to the "unrolling" of the inductive or coinductive type, which is larger than the inductive or coinductive type itself.

The trick is to ensure that the generic extension operation satisfies a *regularity* condition stating that it acts as the identity on constant families, specified as follows:

$$\texttt{map}[\,t\,.\,\tau\,](\,x\,.\,e'\,)(\,e\,) \longmapsto e \qquad (t \notin \tau)$$

In other words the generic extension of a function to a constant type family is the identity.

Exercises

1. Define the generic extension for inductive and coinductive types by giving the following transition rules:

$$\texttt{map}[\,t\,.\,\mu(\,u\,.\,\tau_t^u\,)\,](\,x\,.\,e'\,)(\,e\,) \longmapsto \ldots$$
$$\texttt{map}[\,t\,.\,\nu(\,u\,.\,\tau_t^u\,)\,](\,x\,.\,e'\,)(\,e\,) \longmapsto \ldots$$

2. Why is your solution well-defined? What justifies the validity of your definition?

3. Prove the type preservation property for your stated transition rules, assuming that it holds for the other cases of generic extension given in the main text.

Solutions

1. Let τ_t^u be a polynomial type in two variables, t and u, and write τ_ρ^μ for the substitution instance $\{\rho, \mu/t, u\}\tau$. Let ϕ_t be the type operator $u\,.\,\tau_t^u$, and write ϕ_ρ for $\{\rho/t\}\phi_t$. This expands to $u\,.\,\tau_\rho^u$, and so write $\phi_\rho(\mu)$ for τ_ρ^μ. Let μ_t be the type $\mu(\,u\,.\,\phi_t(u)\,)$, and let ν_t be the type $\nu(\,u\,.\,\phi_t(u)\,)$, so that the type μ_ρ is $\mu(\,u\,.\,\phi_\rho(u)\,)$ and ν_ρ is $\nu(\,u\,.\,\phi_\rho(u)\,)$.

   Suppose that $x : \sigma \vdash e' : \sigma'$. The generic extension for inductive and coinductive types is defined by the following transitions:

$$\texttt{map}[\,t\,.\,\mu(\,u\,.\,\phi_t(u)\,)\,](\,x\,.\,e'\,)(\,e\,) \longmapsto \texttt{rec}[\,u\,.\,\phi_\sigma(u)\,](\,y\,.\,\texttt{fold}[\,u\,.\,\phi_{\sigma'}(u)\,](\,\texttt{map}[\,t\,.\,\phi_t(\mu_{\sigma'})\,](\,x\,.\,e'\,)(\,y\,)\,)\,;e\,)$$

$$\texttt{map}[\,t\,.\,\nu(\,u\,.\,\phi_t(u)\,)\,](\,x\,.\,e'\,)(\,e\,) \longmapsto \texttt{gen}[\,u\,.\,\phi_{\sigma'}(u)\,](\,y\,.\,\texttt{map}[\,t\,.\,\phi_t(\nu_\sigma)\,](\,x\,.\,e'\,)(\,\texttt{unfold}[\,u\,.\,\phi_\sigma(u)\,](\,y\,)\,)\,;e\,)$$

2. The regularity requirement saves the day: the use of generic extension on the right is "larger" than on the left only insofar as occurrences of the type variable u have been replaced by the inductive or coinductive type itself. But a careful inspection of the code reveals that these are closed types, and hence cannot involve the distinguished type variable t. Consequently, regularity ensures that the generic extension to closed types is immaterial, thereby ensuring that the extension is well-defined.

3. Suppose that $x : \sigma \vdash e' : \sigma'$ is the map to be extended. In the inductive case suppose that $e : \mu_\sigma$, and show that the right-hand side has type $\mu_{\sigma'}$. Check that

$$y : \phi_\sigma(\mu_{\sigma'}) \vdash \texttt{map}[\,t\,.\,\phi_t(\mu_{\sigma'})\,](\,x\,.\,e'\,)(\,y\,) : \phi_{\sigma'}(\mu_{\sigma'}),$$

   and so

$$y : \phi_\sigma(\mu_{\sigma'}) \vdash \texttt{fold}[\,u\,.\,\phi_{\sigma'}(u)\,](\,\texttt{map}[\,t\,.\,\phi_t(\mu_{\sigma'})\,](\,x\,.\,e'\,)(\,y\,)\,) : \mu_{\sigma'},$$

   and so the recursor has the required type, $\mu_{\sigma'}$. A similar analysis justifies the coinductive case.

5 Distributed Algol

The distributed programming language **DA** considered in Chapter 41 uses *situated*, or *site-specific*, types to ensure that the expression typing judgment, $e : \tau$,

is *global*, or *unsituated*. For example, the type $\tau\,\mathtt{cmd}[\,w\,]$ classifies expressions that evaluate to encapsulated commands suitable for execution at site w, and the type $\tau\,\mathtt{event}[\,w\,]$ is the type of events that may arise at site w, because of sends or receives along channels allocated at w. Although the expression typing judgment is site-independent, the command typing judgment is not: the judgment $m \div \tau \mathbin{@} w$ states that m is a command returning a value of (situatd) type τ suitable for execution at world w.

An alternative is to consider *global*, or *unsituated* types, which do not tie commands, channels, or events to particular sites, but a *local*, or *situated*, typing judgments $e : \tau \mathbin{@} w$, which state that e is an expression of type τ that may sensibly be evaluated and used at site w. For example, if m is a command that, say, sends along a channel a located at site w, then the the judgment $\mathtt{cmd}(\,m\,) : \mathtt{cmd}(\,\tau\,) \mathbin{@} w$ expresses this fact by putting the site w into the judgment, rather than into the type itself. Typing contexts are similarly relativized to a site, written

$$\Gamma = x_1 : \tau_1 \mathbin{@} w_1, \ldots, x_n : \tau_n \mathbin{@} w_n,$$

to specify the type, as well as site, of the variables $x_1, \ldots, x_n$. Substitution is constrained to allow x_i to be replaced by e_i only if $e_1 : \tau_i \mathbin{@} w$.

The two formulations, which will be explored further below, are linked by the operation $\tau \downarrow w$, which restores ...

To make this precise define $\tau \downarrow w$ to be the absolute type determined by *instantiating* the situated type τ at site w. It is defined by the following illustrative equations:

$$\begin{aligned}
\mathtt{nat} \downarrow w &\triangleq \mathtt{nat} \\
\mathtt{unit} \downarrow w &\triangleq \mathtt{unit} \\
(\tau_1 \times \tau_2) \downarrow w &\triangleq \tau_1 \downarrow w \times \tau_2 \downarrow w \\
(\tau_1 \to \tau_2) \downarrow w &\triangleq \tau_1 \downarrow w \to \tau_2 \downarrow w \\
\tau\,\mathtt{chan} \downarrow w &\triangleq (\tau \downarrow w)\,\mathtt{chan}[\,w\,] \\
\tau\,\mathtt{event} \downarrow w &\triangleq (\tau \downarrow w)\,\mathtt{event}[\,w\,] \\
\tau\,\mathtt{cmd} \downarrow w &\triangleq (\tau \downarrow w)\,\mathtt{cmd}[\,w\,]
\end{aligned}$$

Instantiation situates an unsituated type at a given site. This operation is extended to situated typing contexts $\Psi = x_1 : \tau_1 \mathbin{@} w_1, \ldots, x_n : \tau_n \mathbin{@} w_n$ by defining the unsituated context $\downarrow \Psi$ by the equation

$$\downarrow (x_1 : \tau_1 \mathbin{@} w_1, \ldots, x_n : \tau_n \mathbin{@} w_n) \triangleq x_1 : \tau_1 \downarrow w_1, \ldots, x_n : \tau_n \downarrow w_n.$$

Some types, such as $\mathtt{nat}$, have site-insensitive meaning in that they have the same values regardless of site. (Contrast types such as $\tau\,\mathtt{cmd}$ which classifies commands that may make sense only at a particular site.) An unsituated type, τ, is *mobile* iff for every w and w' the equation $\tau \downarrow w = \tau \downarrow w'$ holds: no matter where it is situated, the result is the same. Such types are important when formulating the statics of **DA** using unsituated types.

The following two exercises must be done together, because each requires the solution to the other. But it is useful nonetheless to separate concerns to the large extent possible.

Exercises

1. Formulate the statics of expressions in **DA** using only unsituated types, but situated typing judgments, such that the following correctness property holds:

   If $\Psi \vdash e : \tau \mathbin{@} w$, then $\downarrow \Psi \vdash e : \tau \downarrow w$.

2. Reformulate the statics of commands in **DA** using only unsituated types, but situated typing judgments, such that the following static correctness property holds:

   If $\Psi \vdash m \div \tau \mathbin{@} w$, then $\downarrow \Psi \vdash m \div \tau \downarrow w \mathbin{@} w$.

Solutions

1.

$$\frac{}{\Psi\, x : \tau \mathbin{@} w \vdash x : \tau \mathbin{@} w} \tag{5a}$$

$$\frac{\Psi \vdash m \div \tau \mathbin{@} w}{\Psi \vdash \mathtt{cmd}(\, m \,) : \mathtt{cmd}(\, \tau \,) \mathbin{@} w} \tag{5b}$$

2.

$$\frac{\Psi \vdash e : \tau \mathbin{@} w}{\Psi \vdash \mathtt{ret}(\, e \,) \div \tau \mathbin{@} w} \tag{6a}$$

$$\frac{\Psi \vdash e : \mathtt{cmd}(\, \tau \,) \mathbin{@} w \quad \Psi, x : \tau \mathbin{@} w \vdash m \div \tau' \mathbin{@} w}{\Psi \vdash \mathtt{bnd}(\, e \,;\, x \,.\, m \,) \div \tau' \mathbin{@} w} \tag{6b}$$

$$\frac{\Psi \vdash m' \div \tau' \mathbin{@} w' \quad \tau' \text{ mobile}}{\Psi \vdash \mathtt{at}[\, w' \,](\, m' \,) \div \tau' \mathbin{@} w} \tag{6c}$$

The requirement that the type τ' in Rule (6c) be mobile is of the essence! It ensures that $\tau' \downarrow w' = \tau' \downarrow w$, which is required for the returned value to make sense at the originating site.