CS 430/530
Formal Semantics

Zhong Shao

Yale University
Department of Computer Science

Infinite Data Types

March 25, 2025

Type-Generic Programming

Consider a function f of type ρ → ρ′, which transforms values of type ρ into values of type ρ′. For example, f might be the doubling function on natural numbers.

Extend f to a transformation from type [ρ/t]τ to type [ρ′/t]τ by applying f to various spots in the input, where a value of type ρ occurs to obtain a value of type ρ′, leaving the rest of the data structure alone.

For example, τ might be bool × t, in which case f could be extended to a function of type bool × ρ → bool × ρ′ that sends the pairs ⟨a, b⟩ to the pair ⟨a, f (b)⟩.

We need a template that marks the occurrences of t in τ at which f is applied. Such a template is known as a *type operator*, t.τ, which is an abstractor binding a type variable t within a type τ.

Polynomial Type Operators

A *type operator* is a type equipped with a designated variable whose occurrences mark the spots in the type where a transformation is applied. A type operator is an abstractor $t.\tau$ such that t type $\vdash \tau$ type. An example of a type operator is the abstractor

$$t.\mathtt{unit} + (\mathtt{bool} \times t)$$

The *polynomial* type operators are those constructed from the type variable t the types `void` and `unit`, and the product and sum type constructors $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$. More precisely, the judgment $t.\tau$ poly is inductively defined by the following rules:

$$\frac{}{t.t \text{ poly}} \tag{14.1a}$$

$$\frac{}{t.\mathtt{unit} \text{ poly}} \tag{14.1b}$$

$$\frac{t.\tau_1 \text{ poly} \quad t.\tau_2 \text{ poly}}{t.\tau_1 \times \tau_2 \text{ poly}} \tag{14.1c}$$

$$\frac{}{t.\mathtt{void} \text{ poly}} \tag{14.1d}$$

$$\frac{t.\tau_1 \text{ poly} \quad t.\tau_2 \text{ poly}}{t.\tau_1 + \tau_2 \text{ poly}} \tag{14.1e}$$

Type-Generic Extension

The *generic extension* of a polynomial type operator is a form of expression with the following syntax

$$\mathsf{Exp}\quad e\quad ::=\quad \texttt{map}\{t.\tau\}(x.e')(e)\quad \texttt{map}\{t.\tau\}(x.e')(e)\quad \text{generic extension.}$$

Its statics is given as follows:

$$\frac{t.\tau\ \mathsf{poly}\quad \Gamma, x:\rho \vdash e':\rho'\quad \Gamma \vdash e:[\rho/t]\tau}{\Gamma \vdash \texttt{map}\{t.\tau\}(x.e')(e):[\rho'/t]\tau} \tag{14.2}$$

The abstractor $x.e'$ specifies a mapping that sends $x:\rho$ to $e':\rho'$. The generic extension of $t.\tau$ along $x.e'$ specifies a mapping from $[\rho/t]\tau$ to $[\rho'/t]\tau$. The latter mapping replaces values v of type ρ occurring at spots corresponding to occurrences of t in τ by the transformed value $[v/x]e'$ of type ρ' at the same spot. The type operator $t.\tau$ is a template in which certain spots, marked by occurrences of t, show where to apply the transformation $x.e'$ to a value of type $[\rho/t]\tau$ to obtain a value of type $[\rho'/t]\tau$.

Type-Generic Extension

The following dynamics makes precise the concept of the generic extension of a polynomial type operator.

$$\frac{}{\mathtt{map}\{t.t\}(x.e')(e) \longmapsto [e/x]e'} \tag{14.3a}$$

$$\frac{}{\mathtt{map}\{t.\mathtt{unit}\}(x.e')(e) \longmapsto e} \tag{14.3b}$$

$$\begin{array}{c} \mathtt{map}\{t.\tau_1 \times \tau_2\}(x.e')(e) \\ \longmapsto \\ \langle \mathtt{map}\{t.\tau_1\}(x.e')(e \cdot \mathtt{l}), \mathtt{map}\{t.\tau_2\}(x.e')(e \cdot \mathtt{r})\rangle \end{array} \tag{14.3c}$$

$$\frac{}{\mathtt{map}\{t.\mathtt{void}\}(x.e')(e) \longmapsto \mathtt{abort}(e)} \tag{14.3d}$$

$$\begin{array}{c} \mathtt{map}\{t.\tau_1 + \tau_2\}(x.e')(e) \\ \longmapsto \\ \mathtt{case}\, e\, \{\mathtt{l} \cdot x_1 \hookrightarrow \mathtt{l} \cdot \mathtt{map}\{t.\tau_1\}(x.e')(x_1) \mid \mathtt{r} \cdot x_2 \hookrightarrow \mathtt{r} \cdot \mathtt{map}\{t.\tau_2\}(x.e')(x_2)\} \end{array} \tag{14.3e}$$

Type-Generic Extension

Theorem 14.1 (Preservation)**.** *If* $\texttt{map}\{t.\tau\}(x.e')(e) : \tau'$ *and* $\texttt{map}\{t.\tau\}(x.e')(e) \longmapsto e''$*, then* $e'' : \tau'$.

Proof By inversion of rule (14.2), we have

1. t type $\vdash \tau$ type;
2. $x : \rho \vdash e' : \rho'$ for some ρ and ρ';
3. $e : [\rho/t]\tau$;
4. τ' is $[\rho'/t]\tau$.

The proof proceeds by cases on rules (14.3). For example, consider rule (14.3c). It follows from inversion that $\texttt{map}\{t.\tau_1\}(x.e')(e \cdot \texttt{l}) : [\rho'/t]\tau_1$, and similarly that $\texttt{map}\{t.\tau_2\}(x.e')(e \cdot \texttt{r}) : [\rho'/t]\tau_2$. It is easy to check that

$$\langle \texttt{map}\{t.\tau_1\}(x.e')(e \cdot \texttt{l}), \texttt{map}\{t.\tau_2\}(x.e')(e \cdot \texttt{r}) \rangle$$

has type $[\rho'/t](\tau_1 \times \tau_2)$, as required. □

Positive Type Operators

We define the judgment $t.\tau$ pos, which states that the abstractor $t.\tau$ is a positive type operator by the following rules:

$$\frac{}{t.t \text{ pos}} \tag{14.4a}$$

$$\frac{}{t.\texttt{unit} \text{ pos}} \tag{14.4b}$$

$$\frac{t.\tau_1 \text{ pos} \quad t.\tau_2 \text{ pos}}{t.\tau_1 \times \tau_2 \text{ pos}} \tag{14.4c}$$

$$\frac{}{t.\texttt{void} \text{ pos}} \tag{14.4d}$$

$$\frac{t.\tau_1 \text{ pos} \quad t.\tau_2 \text{ pos}}{t.\tau_1 + \tau_2 \text{ pos}} \tag{14.4e}$$

$$\frac{\tau_1 \text{ type} \quad t.\tau_2 \text{ pos}}{t.\tau_1 \to \tau_2 \text{ pos}} \tag{14.4f}$$

Positive Type Operators

The generic extension of a positive type operator is defined similarly to that of a polynomial type operator, with the following dynamics on function types:

$$\frac{}{\texttt{map}^+\{t.\tau_1 \to \tau_2\}(x.e')(e) \longmapsto \lambda\,(x_1 : \tau_1)\,\texttt{map}^+\{t.\tau_2\}(x.e')(e(x_1))} \tag{14.5}$$

Because t is not allowed to occur within the domain type, the type of the result is $\tau_1 \to [\rho'/t]\tau_2$, assuming that e is of type $\tau_1 \to [\rho/t]\tau_2$. It is easy to verify preservation for the generic extension of a positive type operator.

Inductive & Coinductive Types

Two important forms of recursive type

- Inductive types: *least*, or *initial*, solutions of certain type equations
- coinductive types: *greatest*, or *final*, solutions of certain type equations

The elements of an inductive type

- a finite composition of its **introduction** forms.
- If we specify the behavior of a function on each of the introduction forms of an inductive type, then its behavior is defined for all values of that type. Such a function is a ***recursor***, or *catamorphism*.

The elements of a coinductive type

- a finite composition of its **elimination** forms.
- If we specify the behavior of an element on each elimination form, then we have fully specified a value of that type. Such an element is a ***generator***, or *anamorphism*.

Nullary and Binary Products

The abstract syntax of products is given by the following grammar:

Typ	τ	$::=$	$\mathtt{unit}$	$\mathtt{unit}$	nullary product
			$\mathtt{prod}(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$	binary product
Exp	e	$::=$	$\mathtt{triv}$	$\langle\rangle$	null tuple
			$\mathtt{pair}(e_1;e_2)$	$\langle e_1, e_2\rangle$	ordered pair
			$\mathtt{pr[l]}(e)$	$e \cdot \mathtt{l}$	left projection
			$\mathtt{pr[r]}(e)$	$e \cdot \mathtt{r}$	right projection

The statics of product types is given by the following rules.

$$\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit}} \tag{10.1a}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2} \tag{10.1b}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{l} : \tau_1} \tag{10.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \cdot \mathtt{r} : \tau_2} \tag{10.1d}$$

Nullary and Binary Sums

The abstract syntax of sums is given by the following grammar:

Typ	τ	$::=$	$\mathtt{void}$	$\mathtt{void}$	nullary sum
			$\mathtt{sum}(\tau_1;\tau_2)$	$\tau_1 + \tau_2$	binary sum
Exp	e	$::=$	$\mathtt{abort}\{\tau\}(e)$	$\mathtt{abort}(e)$	abort
			$\mathtt{in}[\mathtt{l}]\{\tau_1;\tau_2\}(e)$	$\mathtt{l}\cdot e$	left injection
			$\mathtt{in}[\mathtt{r}]\{\tau_1;\tau_2\}(e)$	$\mathtt{r}\cdot e$	right injection
			$\mathtt{case}(e;x_1.e_1;x_2.e_2)$	$\mathtt{case}\,e\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\}$	case analysis

The nullary sum represents a choice of zero alternatives, and hence admits no introduction form. The elimination form, $\mathtt{abort}(e)$, aborts the computation in the event that e evaluates to a value, which it cannot do. The elements of the binary sum type are labeled to show whether they are drawn from the left or the right summand, either $\mathtt{l}\cdot e$ or $\mathtt{r}\cdot e$. A value of the sum type is eliminated by case analysis.

The statics of sum types is given by the following rules.

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{abort}(e) : \tau} \tag{11.1a}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{l}\cdot e : \tau_1 + \tau_2} \tag{11.1b}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{r}\cdot e : \tau_1 + \tau_2} \tag{11.1c}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}\,e\,\{\mathtt{l}\cdot x_1 \hookrightarrow e_1 \mid \mathtt{r}\cdot x_2 \hookrightarrow e_2\} : \tau} \tag{11.1d}$$

Inductive Type Example: Nat

With a view towards deriving the type $\mathtt{nat}$ as a special case of an inductive type, it is useful to combine zero and successor into a single introduction form, and to correspondingly combine the basis and inductive step of the iterator. The following rules specify the statics of this reformulation:

$$\frac{\Gamma \vdash e : \mathtt{unit} + \mathtt{nat}}{\Gamma \vdash \mathtt{fold}_{\mathtt{nat}}(e) : \mathtt{nat}} \tag{15.1a}$$

$$\frac{\Gamma, x : \mathtt{unit} + \tau \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathtt{nat}}{\Gamma \vdash \mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2) : \tau} \tag{15.1b}$$

The expression $\mathtt{fold}_{\mathtt{nat}}(e)$ is the unique introduction form of the type $\mathtt{nat}$. Using this, the expression $\mathtt{z}$ is $\mathtt{fold}_{\mathtt{nat}}(\mathtt{l} \cdot \langle\rangle)$, and $\mathtt{s}(e)$ is $\mathtt{fold}_{\mathtt{nat}}(\mathtt{r} \cdot e)$. The recursor, $\mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2)$, takes as argument the abstractor $x.e_1$ that combines the basis and inductive step into a single computation that, given a value of type $\mathtt{unit} + \tau$, yields a value of type τ. Intuitively, if x is replaced by the value $\mathtt{l} \cdot \langle\rangle$, then e_1 computes the base case of the recursion, and if x is replaced by the value $\mathtt{r} \cdot e$, then e_1 computes the inductive step from the result e of the recursive call.

Inductive Type Example: Nat

$$\frac{}{\mathtt{fold}_{\mathtt{nat}}(e)\ \mathsf{val}} \tag{15.2a}$$

$$\frac{e_2 \longmapsto e_2'}{\mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2) \longmapsto \mathtt{rec}_{\mathtt{nat}}(x.e_1; e_2')} \tag{15.2b}$$

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}(x.e_1; \mathtt{fold}_{\mathtt{nat}}(e_2)) \\ \longmapsto \\ [\mathtt{map}\{t.\mathtt{unit} + t\}(y.\mathtt{rec}_{\mathtt{nat}}(x.e_1; y))(e_2)/x]e_1\end{array}} \tag{15.2c}$$

Rule (15.2c) uses (polynomial) generic extension (see Chapter 14) to apply the recursor to the predecessor, if any, of a natural number. If we expand the definition of the generic extension in place, we obtain this rule:

$$\frac{}{\begin{array}{c}\mathtt{rec}_{\mathtt{nat}}(x.e_1; \mathtt{fold}_{\mathtt{nat}}(e_2)) \\ \longmapsto \\ [\mathtt{case}\, e_2\, \{\mathtt{l} \cdot \_ \hookrightarrow \mathtt{l} \cdot \langle\rangle \mid \mathtt{r} \cdot y \hookrightarrow \mathtt{r} \cdot \mathtt{rec}_{\mathtt{nat}}(x.e_1; y)\}/x]e_1\end{array}}$$

Coinductive Type Example: Stream of Nat

A stream is given by its behavior under the elimination forms for the stream type: $\texttt{hd}(e)$ returns the next, or head, element of the stream, and $\texttt{tl}(e)$ returns the tail of the stream, the stream resulting when the head element is removed. A stream is introduced by a *generator*, the dual of a recursor, that defines the head and the tail of the stream in terms of the current state of the stream, which is represented by a value of some type. The statics of streams is given by the following rules:

$$\frac{\Gamma \vdash e : \texttt{stream}}{\Gamma \vdash \texttt{hd}(e) : \texttt{nat}} \tag{15.3a}$$

$$\frac{\Gamma \vdash e : \texttt{stream}}{\Gamma \vdash \texttt{tl}(e) : \texttt{stream}} \tag{15.3b}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e_1 : \texttt{nat} \quad \Gamma, x : \tau \vdash e_2 : \tau}{\Gamma \vdash \texttt{strgen}\, x\, \texttt{is}\, e\, \texttt{in} <\texttt{hd} \hookrightarrow e_1, \texttt{tl} \hookrightarrow e_2> : \texttt{stream}} \tag{15.3c}$$

Coinductive Type Example: Stream of Nat

The dynamics of streams is given by the following rules:

$$\frac{}{\texttt{strgen}\,x\,\texttt{is}\,e\,\texttt{in}\,\langle\texttt{hd} \hookrightarrow e_1, \texttt{tl} \hookrightarrow e_2\rangle\ \mathsf{val}} \tag{15.4a}$$

$$\frac{e \longmapsto e'}{\texttt{hd}(e) \longmapsto \texttt{hd}(e')} \tag{15.4b}$$

$$\frac{}{\texttt{hd}(\texttt{strgen}\,x\,\texttt{is}\,e\,\texttt{in}\,\langle\texttt{hd} \hookrightarrow e_1, \texttt{tl} \hookrightarrow e_2\rangle) \longmapsto [e/x]e_1} \tag{15.4c}$$

$$\frac{e \longmapsto e'}{\texttt{tl}(e) \longmapsto \texttt{tl}(e')} \tag{15.4d}$$

$$\frac{}{\begin{array}{c}\texttt{tl}(\texttt{strgen}\,x\,\texttt{is}\,e\,\texttt{in}\,\langle\texttt{hd} \hookrightarrow e_1, \texttt{tl} \hookrightarrow e_2\rangle) \\ \longmapsto \\ \texttt{strgen}\,x\,\texttt{is}\,[e/x]e_2\,\texttt{in}\,\langle\texttt{hd} \hookrightarrow e_1, \texttt{tl} \hookrightarrow e_2\rangle\end{array}} \tag{15.4e}$$

Coinductive Type Example: Stream of Nat

To derive streams as a special case of a coinductive type, we combine the head and the tail into a single elimination form, and reorganize the generator correspondingly. Thus, we consider the following statics:

$$\frac{\Gamma \vdash e : \mathtt{stream}}{\Gamma \vdash \mathtt{unfold}_{\mathtt{stream}}(e) : \mathtt{nat} \times \mathtt{stream}} \tag{15.5a}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \mathtt{nat} \times \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{gen}_{\mathtt{stream}}(x.e_1; e_2) : \mathtt{stream}} \tag{15.5b}$$

The dynamics of streams is given by the following rules:

$$\frac{}{\mathtt{gen}_{\mathtt{stream}}(x.e_1; e_2)\ \mathsf{val}} \tag{15.6a}$$

Coinductive Type Example: Stream of Nat

$$\frac{e \longmapsto e'}{\mathtt{unfold}_{\mathtt{stream}}(e) \longmapsto \mathtt{unfold}_{\mathtt{stream}}(e')} \tag{15.6b}$$

$$\frac{}{\begin{array}{c}\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}(x.e_1; e_2)) \\ \longmapsto \\ \mathtt{map}\{t.\mathtt{nat} \times t\}(y.\mathtt{gen}_{\mathtt{stream}}(x.e_1; y))([e_2/x]e_1)\end{array}} \tag{15.6c}$$

Rule (15.6c) uses generic extension to generate a new stream whose state is the second component of $[e_2/x]e_1$. Expanding the generic extension we obtain the following reformulation of this rule:

$$\frac{}{\begin{array}{c}\mathtt{unfold}_{\mathtt{stream}}(\mathtt{gen}_{\mathtt{stream}}(x.e_1; e_2)) \\ \longmapsto \\ \langle([e_2/x]e_1) \cdot \mathtt{l}, \mathtt{gen}_{\mathtt{stream}}(x.e_1; ([e_2/x]e_1) \cdot \mathtt{r})\rangle\end{array}}$$

M: T with Inductive/Coinductive Types

Typ	τ	$::=$	t	t	self-reference
			$\mathtt{ind}(t.\tau)$	$\mu(t.\tau)$	inductive
			$\mathtt{coi}(t.\tau)$	$\nu(t.\tau)$	coinductive

Type formation judgments have the form

$$t_1 \text{ type}, \ldots, t_n \text{ type} \vdash \tau \text{ type},$$

$$\frac{}{\Delta, t \text{ type} \vdash t \text{ type}}$$

$$\frac{}{\Delta \vdash \mathtt{unit} \text{ type}}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{prod}(\tau_1; \tau_2) \text{ type}}$$

$$\frac{}{\Delta \vdash \mathtt{void} \text{ type}}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{sum}(\tau_1; \tau_2) \text{ type}}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \mathtt{arr}(\tau_1; \tau_2) \text{ type}}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash t.\tau \text{ pos}}{\Delta \vdash \mathtt{ind}(t.\tau) \text{ type}}$$

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type} \quad \Delta \vdash t.\tau \text{ pos}}{\Delta \vdash \mathtt{coi}(t.\tau) \text{ type}}$$

Language M: Syntax & Statics

The abstract syntax of **M** is given by the following grammar:

Exp	e	$::=$	$\texttt{fold}\{t.\tau\}(e)$	$\texttt{fold}_{t.\tau}(e)$	constructor
			$\texttt{rec}\{t.\tau\}(x.e_1; e_2)$	$\texttt{rec}(x.e_1; e_2)$	recursor
			$\texttt{unfold}\{t.\tau\}(e)$	$\texttt{unfold}_{t.\tau}(e)$	destructor
			$\texttt{gen}\{t.\tau\}(x.e_1; e_2)$	$\texttt{gen}(x.e_1; e_2)$	generator

The statics for **M** is given by the following typing rules:

$$\frac{\Gamma \vdash e : [\texttt{ind}(t.\tau)/t]\tau}{\Gamma \vdash \texttt{fold}\{t.\tau\}(e) : \texttt{ind}(t.\tau)} \tag{15.8a}$$

$$\frac{\Gamma, x : [\tau'/t]\tau \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \texttt{ind}(t.\tau)}{\Gamma \vdash \texttt{rec}\{t.\tau\}(x.e_1; e_2) : \tau'} \tag{15.8b}$$

$$\frac{\Gamma \vdash e : \texttt{coi}(t.\tau)}{\Gamma \vdash \texttt{unfold}\{t.\tau\}(e) : [\texttt{coi}(t.\tau)/t]\tau} \tag{15.8c}$$

$$\frac{\Gamma \vdash e_2 : \tau_2 \quad \Gamma, x : \tau_2 \vdash e_1 : [\tau_2/t]\tau}{\Gamma \vdash \texttt{gen}\{t.\tau\}(x.e_1; e_2) : \texttt{coi}(t.\tau)} \tag{15.8d}$$

Language M: Dynamics

$$\frac{}{\texttt{fold}\{t.\tau\}(e)\ \mathsf{val}} \tag{15.9a}$$

$$\frac{e_2 \longmapsto e_2'}{\texttt{rec}\{t.\tau\}(x.e_1; e_2) \longmapsto \texttt{rec}\{t.\tau\}(x.e_1; e_2')} \tag{15.9b}$$

$$\frac{}{\begin{array}{c}\texttt{rec}\{t.\tau\}(x.e_1; \texttt{fold}\{t.\tau\}(e_2)) \\ \longmapsto \\ [\texttt{map}^+\{t.\tau\}(y.\texttt{rec}\{t.\tau\}(x.e_1; y))(e_2)/x]e_1\end{array}} \tag{15.9c}$$

$$\frac{}{\texttt{gen}\{t.\tau\}(x.e_1; e_2)\ \mathsf{val}} \tag{15.9d}$$

$$\frac{e \longmapsto e'}{\texttt{unfold}\{t.\tau\}(e) \longmapsto \texttt{unfold}\{t.\tau\}(e')} \tag{15.9e}$$

$$\frac{}{\begin{array}{c}\texttt{unfold}\{t.\tau\}(\texttt{gen}\{t.\tau\}(x.e_1; e_2)) \\ \longmapsto \\ \texttt{map}^+\{t.\tau\}(y.\texttt{gen}\{t.\tau\}(x.e_1; y))([e_2/x]e_1)\end{array}} \tag{15.9f}$$

Language M

Lemma 15.1. *If $e : \tau$ and $e \longmapsto e'$, then $e' : \tau$.*

Proof By rule induction on rules (15.9). □

Lemma 15.2. *If $e : \tau$, then either e val or there exists e' such that $e \longmapsto e'$.*

Proof By rule induction on rules (15.8). □

Although a proof of this fact lies beyond our current reach, all programs in **M** terminate.

Theorem 15.3 (Termination for **M**)**.** *If $e : \tau$, then there exists e' val such that $e \longmapsto^* e'$.*

Solving Type Equations

For a positive type operator $t.\tau$, we may say that the inductive type $\mu(t.\tau)$ and the coinductive type $\nu(t.\tau)$ are both *solutions* (up to isomorphism) of the type equation $t \cong \tau$:

$$\mu(t.\tau) \cong [\mu(t.\tau)/t]\tau$$
$$\nu(t.\tau) \cong [\nu(t.\tau)/t]\tau.$$

Whereas both are solutions to the same type equation, they are not isomorphic to each other. To see why, consider the inductive type $\texttt{nat} \triangleq \mu(t.\texttt{unit} + t)$ and the coinductive type $\texttt{conat} \triangleq \nu(t.\texttt{unit} + t)$. Informally, `nat` is the smallest (most restrictive) type containing zero, given by $\texttt{fold}(\texttt{l} \cdot \langle\rangle)$, and closed under formation of the successor of any other e of type `nat`, given by $\texttt{fold}(\texttt{r} \cdot e)$. Dually, `conat` is the largest (most permissive) type of expressions e for which the unfolding, $\texttt{unfold}(e)$, is either zero, given by $\texttt{l} \cdot \langle\rangle$, or to the successor of some other e' of type `conat`, given by $\texttt{r} \cdot e'$.

Solving Type Equations

Because `nat` is defined by the composition of its introduction forms and sum injections, it is clear that only finite natural numbers can be constructed in finite time. Because `conat` is defined by the composition of its elimination forms (unfoldings plus case analyses), it is clear that a co-natural number can only be explored to finite depth in finite time—essentially we can only examine some finite number of predecessors of a given co-natural number in a terminating program. Consequently,

1. there is a function $i : \texttt{nat} \to \texttt{conat}$ embedding every finite natural number into the type of possibly infinite natural numbers; and
2. there is an "actually infinite" co-natural number ω that is essentially an infinite composition of successors.

Defining the embedding of `nat` into `conat` is the subject of Exercise **15.1**. The infinite co-natural number ω is defined as follows:

$$\omega \triangleq \texttt{gen}(x.\texttt{r} \cdot x; \langle\rangle).$$