Chapter 5

Relational algebraic ornamentation

This chapter turns to the **synthetic** direction of the interconnection between internalism and externalism. As stated in **??**, internalist types can be hard to read and write, and it would be helpful to be able to switch to an alternative language for understanding and deriving internalist types. The alternative language adopted in this chapter is the **relational** language (Section 5.1), of which Bird and de Moor [1997] gave an authoritative account. Unlike the datatype declaration language, using relations we can give concise yet computationally intuitive specifications, which are amenable to manipulation by algebraic laws and theorems. A particularly expressive relational construction is **relational folds**, and when fixing a basic datatype and casting a relational fold as the externalist predicate, we can synthesise a corresponding internalist datatype on the other side of the conversion isomorphism. More specifically, every relational algebra gives rise to an **algebraic ornamentation** (Section 5.2), whose optimised predicate (**??**) can be swapped (**??**) for the relational fold with the algebra. Specifications involving relational folds can then be met by constructing internalist programs whose types involve corresponding algebraically ornamented datatypes. Several examples are given in Section 5.3, followed by some discussion in Section 5.4.

5.1 Relational programming in Agda

Functional programs are known for their amenability to algebraic calculation (see, e.g., Backus [1978] and Bird [2010]), leading to one form of program correctness by construction: one begins with a specification in the form of a functional program that expresses a straightforward but possibly inefficient computation, and transforms it into an extensionally equal but more efficient functional program by applying algebraic laws and theorems. Using functional programs as the specification language means that specifications are directly executable, but the deterministic nature of functional programs can result in less flexible specifications. For example, when specifying an optimisation problem using a functional program that generates all feasible solutions and chooses an optimal one among them, the program necessarily enforces a particular way of choosing the optimal solution, but such enforcement should not in general be part of the specification. To gain more flexibility, the specification language was later generalised to **relational programs** (see, e.g., Bird [1996]). With relational programs, we specify only the relationship between input and output without actually specifying a way to execute the programs, so specifications in the form of relational programs can be as flexible as possible. Though lacking a directly executable semantics, most relational programs can still be read computationally as potentially partial and nondeterministic mappings, so relational specifications largely remain computationally intuitive as functional specifications.

To emphasise the computational interpretation of relations, we will mainly model a relation between sets A and B as a function sending each inhabitant of A to a subset of B. We define subsets by

$$\begin{aligned}&\mathcal{P} \;:\; \mathsf{Set} \to \mathsf{Set}_1\\&\mathcal{P}\,A \;=\; A \to \mathsf{Set}\end{aligned}$$

That is, a subset $s : \mathcal{P}\,A$ is a characteristic function that assigns a type to each inhabitant of A, and $a : A$ is considered to be a member of s if the type $s\;a : \mathsf{Set}$ is inhabited. We may regard $\mathcal{P}\,A$ as the type of computations that nondeterministically produce an inhabitant of A. A simple example is

$$\begin{aligned}&\mathit{any} : \{A : \mathsf{Set}\} \to \mathcal{P}\,A\\&\mathit{any} \;=\; \mathit{const}\;\top\end{aligned}$$

The subset *any* : $\mathcal{P}\,A$ associates the unit type $\top$ with every inhabitant of A. Since $\top$ is inhabited, *any* can produce any inhabitant of A. While $\mathcal{P}$ cannot be made into a conventional monad [Moggi, 1991; Wadler, 1992] because it is not an endofunctor, it can still be equipped with the usual monadic programming combinators (giving rise to a "relative monad" [Altenkirch et al., 2010]):

- The monadic unit is defined as

  $$\begin{aligned}&\mathit{return} \;:\; \{A : \mathsf{Set}\} \to A \to \mathcal{P}\,A\\&\mathit{return} \;=\; \_\!\equiv\!\_\end{aligned}$$

  The subset *return* a : $\mathcal{P}\,A$ for some a : A simplifies to $\lambda\, a' \mapsto a \equiv a'$, so a is the only member of the subset.

- The monadic bind is defined as

  $$\begin{aligned}&\_\!\gg\!=\!\_ \;:\; \{A\;B : \mathsf{Set}\} \to \mathcal{P}\,A \to (A \to \mathcal{P}\,B) \to \mathcal{P}\,B\\&\_\!\gg\!=\!\_\;\{A\}\;s\,f \;=\; \lambda\, b \mapsto \Sigma[\,a : A\,]\;\, s\;a \times f\;a\;b\end{aligned}$$

  If s : $\mathcal{P}\,A$ and f : $A \to \mathcal{P}\,B$, then the subset $s \gg\!= f$: $\mathcal{P}\,B$ is the disjoint union of all the subsets $f\;a$: $\mathcal{P}\,B$ where a ranges over the inhabitants of A that belong to s; that is, an inhabitant b : B is a member of $s \gg\!= f$ exactly when there exists some a : A belonging to s such that b is a member of $f\;a$.

(We omit the proofs that the two combinators satisfy the (relative) monad laws up to pointwise isomorphism.) On top of *return* and $\_\!\gg\!=\!\_$, the functorial map on $\mathcal{P}$ is defined as

$$\begin{aligned}&\_\langle\$\rangle \;:\; \{A\;B : \mathsf{Set}\} \to (A \to B) \to \mathcal{P}\,A \to \mathcal{P}\,B\\&f\;\langle\$\rangle\;s \;=\; s \gg\!= \lambda\, a \mapsto \mathit{return}\;(f\;a)\end{aligned}$$

and we also define a two-argument version for convenience:

$$\begin{aligned}&\_\langle\$\rangle^2 \;:\; \{A\;B\;C : \mathsf{Set}\} \to (A \to B \to C) \to \mathcal{P}\,A \to \mathcal{P}\,B \to \mathcal{P}\,C\\&f\;\langle\$\rangle^2\;s\;t \;=\; s \gg\!= \lambda\, a \mapsto t \gg\!= \lambda\, b \mapsto \mathit{return}\;(f\;a\;b)\end{aligned}$$

(The notation is a reference to applicative functors [McBride and Paterson, 2008], allowing us to think of functorial maps of $\mathcal{P}$ as applications of pure functions to effectful arguments.)

We define a relation between two families of sets as a family of relations between corresponding sets in the families:

$$\_{\leadsto}\_ \,:\, \{I \,:\, \mathsf{Set}\} \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to \mathsf{Set}_1$$
$$\_{\leadsto}\_ \;\{I\}\; X\; Y \;=\; \{i \,:\, I\} \to X\; i \to \mathscr{P}\,(Y\; i)$$

which is the usual generalisation of $\_{\rightrightarrows}\_$ to allow nondeterminacy. Below we define several relational operators that we will need.

- Since functions are deterministic relations, we have the following combinator *fun* that lifts functions to relations using *return*.

  $$\mathit{fun} \,:\, \{I \,:\, \mathsf{Set}\}\; \{X\; Y \,:\, I \to \mathsf{Set}\} \to (X \rightrightarrows Y) \to (X \leadsto Y)$$
  $$\mathit{fun}\; f\; x \;=\; \mathit{return}\; (f\; x)$$

- The identity relation is just the identity function lifted by *fun*.

  $$\mathit{idR} \,:\, \{I \,:\, \mathsf{Set}\}\; \{X \,:\, I \to \mathsf{Set}\} \to (X \leadsto X)$$
  $$\mathit{idR} \;=\; \mathit{fun}\; \mathit{id}$$

- Composition of relations $\_{\bullet}\_$ is easily defined with $\_{\gg\!=}\_$: computing $R \bullet S$ on input x is first computing $S\; x$ and then feeding the result to R.

  $$\_{\bullet}\_ \,:\, \{I \,:\, \mathsf{Set}\}\; \{X\; Y\; Z \,:\, I \to \mathsf{Set}\} \to (Y \leadsto Z) \to (X \leadsto Y) \to (X \leadsto Z)$$
  $$(R \bullet S)\; x \;=\; S\; x \gg\!= R$$

- Some relations do not carry obvious computational meaning, but can still be defined pointwise, like the **meet** of two relations:

  $$\_{\cap}\_ \,:\, \{I \,:\, \mathsf{Set}\}\; \{X\; Y \,:\, I \to \mathsf{Set}\} \to (X \leadsto Y) \to (X \leadsto Y) \to (X \leadsto Y)$$
  $$(R \cap S)\; x\; y \;=\; R\; x\; y \times S\; x\; y$$

- Unlike a function, which distinguishes between input and output, inherently a relation treats its domain and codomain symmetrically. This is reflected by the presence of the following **converse** operator:

  $$\_^{\circ} \,:\, \{I \,:\, \mathsf{Set}\}\; \{X\; Y \,:\, I \to \mathsf{Set}\} \to (X \leadsto Y) \to (Y \leadsto X)$$
  $$(R^{\,\circ})\; y\; x \;=\; R\; x\; y$$

  A relation can thus be “run backwards” simply by taking its converse. The nondeterministic and bidirectional nature of relations makes them a powerful and concise language for specifications, as will be demonstrated in Sections 5.3.2 and 5.3.3.

- We will also need **relators**, i.e., functorial maps on relations:

$$
\begin{array}{l}
\mathit{mapRDR} : \{I : \mathsf{Set}\}\ (D : \mathsf{RDesc}\ I)\ \{X\ Y : I \to \mathsf{Set}\} \to \\
\quad (X \rightsquigarrow Y) \to [\![\, D \,]\!]\ X \to \mathscr{P}([\![\, D \,]\!]\ Y) \\
\mathit{mapRDR}\ (\mathsf{v}\ [])\qquad R\ \blacksquare \qquad\quad = \mathit{return}\ \blacksquare \\
\mathit{mapRDR}\ (\mathsf{v}\ (i :: \mathit{is}))\ R\ (x\ ,\ \mathit{xs})\ = \_,\_\ \langle\$\rangle^2\ (R\ x)\ (\mathit{mapRDR}\ (\mathsf{v}\ \mathit{is})\ R\ \mathit{xs}) \\
\mathit{mapRDR}\ (\sigma\ S\ D)\quad R\ (s\ ,\ \mathit{xs})\ = (\_,\_\ s)\ \langle\$\rangle\ (\mathit{mapRDR}\ (D\ s)\ R\ \mathit{xs}) \\
\mathbb{R} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{X\ Y : I \to \mathsf{Set}\} \to (X \rightsquigarrow Y) \to (\mathbb{F}\ D\ X \rightsquigarrow \mathbb{F}\ D\ Y) \\
\mathbb{R}\ D\ R\ \{i\} = \mathit{mapRDR}\ (D\ i)\ R
\end{array}
$$

Figure 5.1 Definitions for relators.

$$
\begin{array}{l}
\mathbb{R} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{X\ Y : I \to \mathsf{Set}\} \to \\
\qquad (X \rightsquigarrow Y) \to (\mathbb{F}\ D\ X \rightsquigarrow \mathbb{F}\ D\ Y)
\end{array}
$$

If $R : X \rightsquigarrow Y$, the relation $\mathbb{R}\ D\ R : \mathbb{F}\ D\ X \rightsquigarrow \mathbb{F}\ D\ Y$ applies R to the recursive positions of its input, leaving everything else intact. The definition of $\mathbb{R}$ is shown in Figure 5.1. For example, if $D = \mathit{ListD}\ A$, then $\mathbb{R}\ (\mathit{ListD}\ A)$ is, up to isomorphism,

$$
\begin{array}{l}
\mathbb{R}\ (\mathit{ListD}\ A) : \{X\ Y : I \to \mathsf{Set}\} \to \\
\qquad\qquad (X \rightsquigarrow Y) \to (\mathbb{F}\ (\mathit{ListD}\ A)\ X \rightsquigarrow \mathbb{F}\ (\mathit{ListD}\ A)\ Y) \\
\mathbb{R}\ (\mathit{ListD}\ A)\ R\ (\text{‘}\mathsf{nil}\ ,\qquad \blacksquare) = \mathit{return}\ (\text{‘}\mathsf{nil}\ ,\ \blacksquare) \\
\mathbb{R}\ (\mathit{ListD}\ A)\ R\ (\text{‘}\mathsf{cons}\ ,\ a\ ,\ x\ ,\ \blacksquare) = (\lambda\, y \mapsto \text{‘}\mathsf{cons}\ ,\ a\ ,\ y\ ,\ \blacksquare)\ \langle\$\rangle\ (R\ x)
\end{array}
$$

Laws and theorems about relational programs are formulated with relational inclusion:

$$
\begin{array}{l}
\_\subseteq\_ : \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\}\ (X \rightsquigarrow Y) \to (X \rightsquigarrow Y) \to \mathsf{Set} \\
\_\subseteq\_\ \{I\}\ R\ S = \{i : I\}\ (x : X\ i)\ (y : Y\ i) \to R\ x\ y \to S\ x\ y
\end{array}
$$

or equivalence of relations, i.e., two-way inclusion:

$$
\begin{array}{l}
\_\simeq\_ : \{I : \mathsf{Set}\}\ \{X\ Y : I \to \mathsf{Set}\}\ (R\ S : X \rightsquigarrow Y) \to \mathsf{Set} \\
R \simeq S = (R \subseteq S) \times (R \supseteq S)
\end{array}
$$

where $R \supseteq S$ is defined to be $S \subseteq R$ as usual. For example, a relator preserves identity and composition, i.e.,

$$
\mathbb{R}\ D\ \mathit{idR} \simeq \mathit{idR} \qquad \text{and} \qquad \mathbb{R}\ D\ (R \cdot S) \simeq \mathbb{R}\ D\ R \cdot \mathbb{R}\ D\ S
$$

and is monotonic, i.e.,

$$\mathbb{R}\,D\,R \subseteq \mathbb{R}\,D\,S \qquad \text{whenever} \quad R \subseteq S$$

Also, many concepts can be expressed in a surprisingly concise way with relational inclusion. For example, a relation R is a preorder if it is reflexive and transitive. In relational terms, these two conditions are expressed simply as

$$\mathit{idR} \subseteq R \qquad \text{and} \qquad R \bullet R \subseteq R$$

and are easily manipulable in calculations. Another important notion is **monotonic algebras** [Bird and de Moor, 1997, Section 7.2]: an algebra $S : \mathbb{F}\,D\,X \rightsquigarrow X$ is **monotonic** on $R : X \rightsquigarrow X$ (usually an ordering) if

$$S \bullet \mathbb{R}\,D\,R \subseteq R \bullet S$$

which says that if two input values to S have their recursive positions related by R and are otherwise equal, then the output values would still be related by R. (For example, let

$$D = \lambda\,\{\ \blacksquare \mapsto \mathsf{v}\,(\blacksquare :: \blacksquare :: [\,])\ \} : \mathsf{Desc}\ \top$$

be a trivially indexed description with two recursive positions, and define

$$\mathit{plus} = \mathit{fun}\ (\lambda\ \{\ (x\,,y\,,\blacksquare) \mapsto x + y\ \}) : \mathbb{F}\,D\,(\mathit{const}\ \mathsf{Nat}) \rightsquigarrow \mathit{const}\ \mathsf{Nat}$$

Then *plus* is monotonic on

$$\mathit{leq} = \lambda\,x\,y \mapsto y \leqslant x : \mathit{const}\ \mathsf{Nat} \rightsquigarrow \mathit{const}\ \mathsf{Nat}$$

which maps a natural number x to any natural number y that is at most x. Pointwise, the monotonicity statement expands to

$$(x\,y\,x'\,y' : \mathsf{Nat}) \to (x \leqslant x') \times (y \leqslant y') \to x + y \leqslant x' + y'$$

i.e., addition is monotonic on its two arguments.) In the context of optimisation problems, monotonicity can be used to capture the **principle of optimality**, as will be shown in Section 5.3.3. Section 5.3.1 contains some simple relational calculations involving the above properties.

Having defined relations as nondeterministic mappings, it is straightforward to rewrite the datatype-generic *fold* with the subset combinators to obtain a relational version, which is denoted by the "banana bracket" [Meijer et al., 1991]:

mutual

$$
\begin{aligned}
&(\!|\_|\!) : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\ \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \leadsto X) \to (\mu\ D \leadsto X)\\
&(\!|\_|\!)\ \{I\}\ \{D\}\ R\ \{i\}\ (\mathsf{con}\ ds) = \mathit{mapFoldR}\ D\ (D\ i)\ R\ ds \gg\!= R\\
&\mathit{mapFoldR} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ (D' : \mathsf{RDesc}\ I) \to\\
&\qquad \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \leadsto X) \to [\![\, D']\!]\ (\mu\ D) \to \mathscr{P}\,([\![\, D']\!]\ X)\\
&\mathit{mapFoldR}\ D\ (\mathsf{v}\ [])\qquad R\ \blacksquare \qquad = \mathit{return}\ \blacksquare\\
&\mathit{mapFoldR}\ D\ (\mathsf{v}\ (i :: is))\ R\ (d\ , ds) = \_,\_\ \langle\$\rangle^2\ ((\!|\, R\,|\!)\ d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{mapFoldR}\ D\ (\mathsf{v}\ is)\ f\ ds)\\
&\mathit{mapFoldR}\ D\ (\sigma\ S\ D')\quad R\ (s\ , ds) = (\_,\_\ s)\ \langle\$\rangle\ (\mathit{mapFoldR}\ D\ (D'\ s)\ f\ ds)
\end{aligned}
$$

Figure 5.2 Definition of relational folds.

$$(\!|\_|\!) : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\ \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \leadsto X) \to (\mu\ D \leadsto X)$$

The definition of $(\!|\_|\!)$ is shown in Figure 5.2 (cf. the definition of *fold* in **??**). For example, the relational fold on lists is, up to isomorphism,

$$
\begin{aligned}
&(\!|\_|\!)\ \{\top\}\ \{\mathit{ListD}\ A\} : \{X : \top \to \mathsf{Set}\} \to\\
&\qquad\qquad (\mathbb{F}\ (\mathit{ListD}\ A)\ X \leadsto X) \to (\mu\ (\mathit{ListD}\ A) \leadsto X)\\
&(\!|\, R\,|\!)\ [] \qquad = R\ (\text{‘}\mathsf{nil}\ , \blacksquare)\\
&(\!|\, R\,|\!)\ (a :: as) = (\!|\, R\,|\!)\ as \gg\!= \lambda\, x \mapsto R\ (\text{‘}\mathsf{cons}\ , a\ , x\ , \blacksquare)
\end{aligned}
$$

The functional and relational fold operators are related by the following lemma:

$$
\begin{aligned}
\mathit{fun\text{-}preserves\text{-}fold} :\ & \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{X : I \to \mathsf{Set}\} \to\\
& (f : \mathbb{F}\ D\ X \rightrightarrows X)\ \{i : I\}\ (d : \mu\ D\ i)\ (x : X\ i) \to\\
& \mathit{fun}\ (\mathit{fold}\ f)\ d\ x \cong (\!|\, \mathit{fun}\ f\,|\!)\ d\ x
\end{aligned}
$$

which is a strengthened version of $\mathit{fun}\ (\mathit{fold}\ f) \simeq (\!|\, \mathit{fun}\ f\,|\!)$.

5.2 Definition of algebraic ornamentation

We now turn to algebraic ornamentation, the key construct that bridges internalist and relational programming, and look at a special case first. Let

$$R : \mathbb{F}\ (\mathit{ListD}\ A)\ (\mathit{const}\ X) \leadsto \mathit{const}\ X \qquad \text{where} \quad X : \mathsf{Set}$$

be a relational algebra for lists. We can define a datatype of "algebraically ornamented lists" as

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{AlgList}\ A\ R\ :\ X \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{AlgList}\ A\ R\ x\ \ni\ \mathsf{nil}\ (\mathit{rnil}\ :\ R\ ({}^{\backprime}\mathsf{nil}\ ,\ \blacksquare)\ x) \\
\qquad\qquad\qquad\quad \mid\ \mathsf{cons}\ (a\ :\ A)\ (x'\ :\ X)\ (\mathit{rcons}\ :\ R\ ({}^{\backprime}\mathsf{cons}\ ,\ a\ ,\ x'\ ,\ \blacksquare)\ x) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{as}\ :\ \mathsf{AlgList}\ A\ R\ x')
\end{array}
$$

There is an ornament from lists to algebraically ornamented lists which marks the fields *rnil*, x', and *rcons* in AlgList as additional and refines the index of the recursive position from ▪ to x'. The optimised predicate (**??**) for this ornament is

$$
\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{AlgListP}\ A\ R\ :\ X \to \mathsf{List}\ A \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{AlgListP}\ A\ R\ x\ [\,]\qquad\quad\ \ni\ \mathsf{nil}\ (\mathit{rnil}\ :\ R\ ({}^{\backprime}\mathsf{nil}\ ,\ \blacksquare)\ x) \\
\quad \mathsf{AlgListP}\ A\ R\ x\ (a :: \mathit{as})\ \ni\ \mathsf{cons}\ (x'\ :\ X)\ (\mathit{rcons}\ :\ R\ ({}^{\backprime}\mathsf{cons}\ ,\ a\ ,\ x'\ ,\ \blacksquare)\ x) \\
\qquad\qquad\qquad\qquad\qquad\qquad (p\ :\ \mathsf{AlgListP}\ A\ R\ x'\ \mathit{as})
\end{array}
$$

A simple argument by induction shows that AlgListP $A\ R\ x\ \mathit{as}$ is in fact isomorphic to $(\![\, R \,]\!)\ \mathit{as}\ x$ for any $\mathit{as}\ :\ \mathsf{List}\ A$ and $x\ :\ X$ — that is, we can do predicate swapping for the refinement semantics of the ornament from lists to algebraically ornamented lists (**??**). Thus we get the conversion isomorphisms

$$(x\ :\ X) \to \mathsf{AlgList}\ A\ R\ x\ \cong\ \Sigma[\,\mathit{as} : \mathsf{List}\ A\,]\ (\![\, R \,]\!)\ \mathit{as}\ x \tag{5.1}$$

That is, an algebraically ornamented list is exactly a plain list and a proof that the list folds to x using the algebra R. The traditional bottom-up vector datatype is a special case of AlgList: the datatype AlgList A (*fun length-alg*), where *length-alg* is the algebra for *length*, is exactly Vec′ A in **??**. By (5.1) we have

$$(n\ :\ \mathsf{Nat}) \to \mathsf{Vec}'\ A\ n\ \cong\ \Sigma[\,\mathit{as} : \mathsf{List}\ A\,]\ (\![\, \mathit{fun\ length\text{-}alg} \,]\!)\ \mathit{as}\ n$$

from which we can further derive

$$\mathsf{Vec}'\ A\ n\ \cong\ \Sigma[\,\mathit{as} : \mathsf{List}\ A\,]\ \mathit{length\ as}\ \equiv\ n$$

by *fun-preserves-fold*.

The above can be generalised to all datatypes encoded by the Desc universe. Let $D\ :\ \mathsf{Desc}\ I$ be a description and $R\ :\ \mathbb{F}\ D\ X \leadsto X$ (where $X\ :\ I \to \mathsf{Set}$) an algebra. The **algebraic ornamentation** of D with R is an ornamental description

$$\begin{array}{l}
\mathit{algROD} : \{I : \mathsf{Set}\}\ (D : \mathsf{RDesc}\ I)\ \{X : I \to \mathsf{Set}\} \to \\
\qquad\quad \mathscr{P}\ (\llbracket D \rrbracket\ X) \to \mathsf{ROrnDesc}\ (\Sigma\ I\ X)\ \mathit{outl}\ D \\
\mathit{algROD}\ (\mathsf{v}\ \mathit{is}) \quad \{X\}\ P \ =\ \Delta[\mathit{xs} : \mathbb{P}\ \mathit{is}\ X]\ \Delta[\_ : P\ \mathit{xs}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{v}\ (\mathbb{P}\text{-}\mathit{map}\ (\lambda\ \{i\}\ x \mapsto \mathsf{ok}\ (i\,,x))\ \mathit{is}\ \mathit{xs}) \\
\mathit{algROD}\ (\sigma\ S\ D) \qquad P \ =\ \sigma[s : S]\ \mathit{algROD}\ (D\ s)\ (\mathit{curry}\ P\ s) \\
\mathit{algOD} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{X : I \to \mathsf{Set}\} \to \\
\qquad\quad (\mathbb{F}\ D\ X \leadsto X) \to \mathsf{OrnDesc}\ (\Sigma\ I\ X)\ \mathit{outl}\ D \\
\mathit{algOD}\ D\ R\ (\mathsf{ok}\ (i\,,x)) \ =\ \mathit{algROD}\ (D\ i)\ ((R\,^{\circ})\ x)
\end{array}$$

Figure 5.3 Definitions for algebraic ornamentation.

$$\mathit{algOD}\ D\ R : \mathsf{OrnDesc}\ (\Sigma\ I\ X)\ \mathit{outl}\ D$$

(where $\mathit{outl} : \Sigma\ I\ X \to I$). The optimised predicate for $\lceil \mathit{algOD}\ D\ R \rceil$ is pointwise isomorphic to $(\!|\, R \,|\!)$, i.e.,

$$(i : I)\ (x : X\ i)\ (d : \mu\ D\ i) \to \mathsf{OptP}\ \lceil \mathit{algOD}\ D\ R \rceil\ (\mathsf{ok}\ (i\,,x))\ d \cong (\!|\, R \,|\!)\ d\ x$$

which is proved by induction on d. These isomorphisms give rise to a swap family

$$\mathit{algOD\text{-}FSwap}\ D\ R : \mathsf{FSwap}\ (\mathit{RSem}\ \lceil \mathit{algOD}\ D\ R \rceil)$$

such that $\mathsf{Swap}.P\ (\mathit{algOD\text{-}FSwap}\ D\ R\ (\mathsf{ok}\ (i\,,x))) \ =\ \lambda\,d \mapsto (\!|\, R \,|\!)\ d\ x$, so we arrive at the following conversion isomorphisms

$$(i : I)\ (x : X\ i) \to \mu\ \lfloor \mathit{algOD}\ D\ R \rfloor\ (i\,,x) \cong \Sigma[d : \mu\ D\ i]\ (\!|\, R \,|\!)\ d\ x \tag{5.2}$$

AlgList is obtained by defining $\mathsf{AlgList}\ A\ R\ x \ =\ \mu\ \lfloor \mathit{algOD}\ (\mathit{ListD}\ A)\ R \rfloor\ (\bullet\,,x)$. The definition of *algOD*, shown in Figure 5.3, is an adaptation and generalisation of McBride's original definition of functional algebraic ornamentation [2011]. Roughly speaking, it retains (in the σ case of *algROD*) all the fields of the basic description and inserts (in the v case of *algROD*) before every v

- a new field of indices for the recursive positions (e.g., the field x' in AlgList) and
- another new field requesting a proof that
  - the indices supplied in the previous field and

– the values for the fields originally in the basic description
compute to the targeted index through R (e.g., the fields *rnil* and *rcons* in AlgList).

Algebraic ornamentation is a convenient method for adding new indices to inductive families. (We will see in **??** that it is actually a canonical way of refining inductive families by ornamentation.) Most importantly, the conversion isomorphisms (5.2) state clearly what the new indices mean in terms of relations. We can thus easily translate relational expressions into internalist types for type-directed programming, as demonstrated in the next section.

5.3 Examples

We give three examples involving three relational theorems.

- Section 5.3.1 shows how the **Fold Fusion Theorem** [Bird and de Moor, 1997, Section 6.2] gives rise to conversion functions between algebraically ornamented datatypes.
- Section 5.3.2 implements the **Streaming Theorem** [Bird and Gibbons, 2003, Theorem 30] as an internalist program, whose type directly corresponds to the "metamorphic" specification stated by the theorem.
- Section 5.3.3 uses the **Greedy Theorem** [Bird and de Moor, 1997, Theorem 10.1] to nontrivially derive a suitable type for an internalist program that solves the **minimum coin change problem**.

5.3.1 The Fold Fusion Theorem

The statement of the **Fold Fusion Theorem** [Bird and de Moor, 1997, Section 6.2] is as follows: Let $D : \mathsf{Desc}\ I$ be a description, $R : X \rightsquigarrow Y$ a relation, and $S : \mathbb{F}\ D\ X \rightsquigarrow X$ and $T : \mathbb{F}\ D\ Y \rightsquigarrow Y$ algebras. If R is a homomorphism from S to T, i.e.,

$$R \bullet S \simeq T \bullet \mathbb{R}\ D\ R$$

which is referred to as the **fusion condition**, then we have

$$
\begin{aligned}
&\mathit{new}\text{-}\Sigma \;:\; (I : \mathsf{Set})\ \{A : \mathsf{Set}\}\ \{X : I \to \mathsf{Set}\} \to \\
&\qquad ((i : I) \to \mathsf{Upgrade}\ A\ (X\ i)) \to \mathsf{Upgrade}\ A\ (\Sigma\ I\ X) \\
&\mathit{new}\text{-}\Sigma\ I\ u \;=\; \mathbf{record}\ \{\ P = \lambda\, a \mapsto \Sigma[\, i : I\,]\ \mathsf{Upgrade}.P\ (u\ i)\ a \\
&\qquad\qquad\qquad\quad ;\ C = \lambda\ \{\ a\ (i\,,x) \mapsto \mathsf{Upgrade}.C\ (u\ i)\ a\ x\ \ \} \\
&\qquad\qquad\qquad\quad ;\ u = \lambda\ \{\ a\ (i\,,p) \mapsto i\,,\mathsf{Upgrade}.u\ (u\ i)\ a\ p\ \} \\
&\qquad\qquad\qquad\quad ;\ c = \lambda\ \{\ a\ (i\,,p) \mapsto \mathsf{Upgrade}.c\ (u\ i)\ a\ p\ \ \}\ \} \\
&\mathbf{syntax}\ \mathit{new}\text{-}\Sigma\ I\ (\lambda\, i \mapsto u) \;=\; \Sigma^{+}[\, i : I\,]\ u \\
&\_{\times^{+}}\_ \;:\; \{X\ Y : \mathsf{Set}\} \to \mathsf{Upgrade}\ X\ Y \to (Z : \mathsf{Set}) \to \mathsf{Upgrade}\ X\ (Y \times Z) \\
&u \times^{+} Z \;=\; \mathbf{record}\ \{\ P = \lambda\, x \mapsto \mathsf{Upgrade}.P\ u\ x \times Z \\
&\qquad\qquad\quad ;\ C = \lambda\ \{\ x\ (y\,,z) \mapsto \mathsf{Upgrade}.C\ u\ x\ y\ \ \} \\
&\qquad\qquad\quad ;\ u = \lambda\ \{\ x\ (p\,,z) \mapsto \mathsf{Upgrade}.u\ u\ x\ p\,,z\ \} \\
&\qquad\qquad\quad ;\ c = \lambda\ \{\ x\ (p\,,z) \mapsto \mathsf{Upgrade}.c\ u\ x\ p\ \ \}\ \}
\end{aligned}
$$

Figure 5.4 Two additional upgrade combinators.

$$R \bullet (\!|\, S \,|\!) \;\simeq\; (\!|\, T \,|\!)$$

The above is, in fact, a corollary of two variations of Fold Fusion that replace relational equivalence in the statement of the theorem with relational inclusion. One variation is

$$R \bullet S \subseteq T \bullet \mathbb{R}\, D\, R \;\;\to\;\; R \bullet (\!|\, S \,|\!) \subseteq (\!|\, T \,|\!) \tag{5.3}$$

and the other variation simply reverses the direction of inclusion:

$$R \bullet S \supseteq T \bullet \mathbb{R}\, D\, R \;\;\to\;\; R \bullet (\!|\, S \,|\!) \supseteq (\!|\, T \,|\!) \tag{5.4}$$

Both of them roughly state that one fold can be conditionally transformed into another. Since algebraically ornamented datatypes are datatypes with constraints expressed as a fold, we should be able to transform these constraints by the Fold Fusion Theorem while leaving the underlying data unchanged, thus converting one algebraically ornamented datatype into another.

We look at (5.3) first. Assume that we have a proof of the antecedent

$$\mathit{fcond}_{\subseteq} \;:\; R \bullet S \subseteq T \bullet \mathbb{R}\, D\, R$$

Expanding the conclusion of (5.3) pointwise: if $d : \mu\ D\ i$ folds to $x : X\ i$ with S, which is "relaxed" to $y : Y\ i$ by R, then d folds to y with T. This can be translated to a conversion function from a datatype algebraically ornamented with S to one with T:

$$\begin{array}{l} \mathit{fusion\text{-}conversion}_{\subseteq}\ D\ R\ S\ T\ \mathit{fcond}_{\subseteq}\ : \\ \quad \{i : I\}\ (x : X\ i) \rightarrow \mu\ \lfloor \mathit{algOD}\ D\ S \rfloor\ (i\,,x) \rightarrow \\ \quad (y : Y\ i) \rightarrow R\ x\ y \rightarrow \mu\ \lfloor \mathit{algOD}\ D\ T \rfloor\ (i\,,y) \end{array}$$

This function does not alter the underlying $(\mu\ D)$-data, which can be easily expressed by upgrading (**??**) the identity function on $\mu\ D$ to its type. We thus write the following upgrade

$$\begin{array}{l} \mathit{upg}_{\subseteq}\ :\ \mathsf{Upgrade}\ (\{i : I\} \rightarrow \mu\ D\ i \rightarrow \mu\ D\ i) \\ \qquad\qquad\qquad (\{i : I\}\ \{x : X\ i\} \rightarrow \mu\ \lfloor \mathit{algOD}\ D\ S \rfloor\ (i\,,x) \rightarrow \\ \qquad\qquad\qquad\ \ \{y : Y\ i\} \rightarrow R\ x\ y \rightarrow \mu\ \lfloor \mathit{algOD}\ D\ T \rfloor\ (i\,,y)) \\ \mathit{upg}_{\subseteq}\ =\ \forall[[\,i : I\,]]\ \forall^{+}[[\,x : X\ i\,]]\ \mathit{ref\text{-}S}\ i\ x \rightharpoonup \\ \qquad\quad \forall^{+}[[\,y : Y\ i\,]]\ \forall^{+}[\,\_ : R\ x\ y\,]\ \mathit{toUpgrade}\ (\mathit{ref\text{-}T}\ i\ y) \end{array}$$

where the refinements are defined by

$$\begin{array}{l} \mathit{ref\text{-}S}\ :\ (i : I)\ (x : X\ i) \rightarrow \mathsf{Refinement}\ (\mu\ D\ i)\ (\mu\ \lfloor \mathit{algOD}\ D\ S \rfloor\ (i\,,x)) \\ \mathit{ref\text{-}S}\ i\ x\ =\ \mathit{toRefinement}\ (\mathit{algOD\text{-}FSwap}\ D\ S\ (\mathsf{ok}\ (i\,,x))) \\ \mathit{ref\text{-}T}\ :\ (i : I)\ (y : Y\ i) \rightarrow \mathsf{Refinement}\ (\mu\ D\ i)\ (\mu\ \lfloor \mathit{algOD}\ D\ T \rfloor\ (i\,,y)) \\ \mathit{ref\text{-}T}\ i\ y\ =\ \mathit{toRefinement}\ (\mathit{algOD\text{-}FSwap}\ D\ T\ (\mathsf{ok}\ (i\,,y))) \end{array}$$

and implement the conversion function by

$$\mathsf{Upgrade}.u\ \mathit{upg}_{\subseteq}\ \mathit{id}\ \{\ \}_0$$

Goal 0 demands a promotion proof of type

$$\{i : I\}\ \{x : X\ i\}\ (d : \mu\ D\ i) \rightarrow (\![\, S \,]\!)\ d\ x \rightarrow \{y : Y\ i\} \rightarrow R\ x\ y \rightarrow (\![\, T \,]\!)\ (\mathit{id}\ d)\ y$$

which is exactly the pointwise expansion of the conclusion of (5.3). The coherence property

$$\begin{array}{l} \{i : I\}\ \{x : X\ i\}\ (d : \mu\ D\ i)\ (d' : \mu\ \lfloor \mathit{algOD}\ D\ S \rfloor\ (i\,,x)) \rightarrow \\ \quad \mathit{forget}\ \lceil \mathit{algOD}\ D\ S \rceil\ d'\ \equiv\ d \rightarrow \\ \{y : Y\ i\} \rightarrow R\ x\ y \rightarrow \\ \quad \mathit{forget}\ \lceil \mathit{algOD}\ D\ T \rceil\ (\mathit{fusion\text{-}conversion}_{\subseteq}\ D\ R\ S\ T\ \mathit{fcond}_{\subseteq}\ d')\ \equiv\ \mathit{id}\ d \end{array}$$

then states that the conversion function transforms the underlying $(\mu\ D)$-data in the same way as *id*, i.e., it leaves the underlying data unchanged. Similarly for (5.4), assuming that we have

$$\mathit{fcond}_{\supseteq} \;:\; R \bullet S \;\supseteq\; T \bullet \mathbb{R}\, D\, R$$

we should be able to construct the conversion function

$$\begin{aligned}&\mathit{fusion\text{-}conversion}_{\supseteq}\; D\; R\; S\; T\; \mathit{fcond}_{\supseteq} \;:\\ &\quad \{i \,:\, I\}\; (y \,:\, Y\, i) \to \mu\; \lfloor \mathit{algOD}\; D\; T \rfloor\; (i\,,y) \to\\ &\quad \Sigma[\, x : X\, i\,]\;\; \mu\; \lfloor \mathit{algOD}\; D\; S \rfloor\; (i\,,x) \times R\; x\; y\end{aligned}$$

as an upgraded version of the identity function with the upgrade

$$\begin{aligned}\mathit{upg}_{\supseteq} \;:\; &\mathsf{Upgrade}\; (\{i \,:\, I\} \to \mu\; D\; i \to \mu\; D\; i)\\ &\qquad\qquad (\{i \,:\, I\}\; \{y \,:\, Y\, i\} \to \mu\; \lfloor \mathit{algOD}\; D\; T \rfloor\; (i\,,y) \to\\ &\qquad\qquad\; \Sigma[\, x : X\, i\,]\;\; \mu\; \lfloor \mathit{algOD}\; D\; S \rfloor\; (i\,,x) \times R\; x\; y)\\ \mathit{upg}_{\supseteq} \;=\; &\forall^{+}[[\, i : I\,]]\;\; \forall^{+}[[\, y : Y\, i\,]]\;\; \mathit{ref\text{-}T}\; i\; y \rightharpoonup\\ &\;\Sigma^{+}[\, x : X\, i\,]\;\; \mathit{toUpgrade}\; (\mathit{ref\text{-}S}\; i\; x) \times^{+} R\; x\; y\end{aligned}$$

in which we need two new combinators to deal with upgrading to product types, which are defined in Figure 5.4.

For a simple application, suppose that we need a "bounded" vector datatype, i.e., lists indexed with an upper bound on their length. A quick thought might lead to this definition

$$\begin{aligned}&\mathsf{BVec} \;:\; \mathsf{Set} \to \mathsf{Nat} \to \mathsf{Set}\\ &\mathsf{BVec}\; A\; m \;=\; \mu\; \lfloor \mathit{algOD}\; (\mathit{ListD}\; A)\; (\mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg}) \rfloor\; (\blacksquare\,, m)\end{aligned}$$

where $\mathit{geq} \;=\; \mathit{leq}^{\,\circ} \;:\; \mathit{const}\;\mathsf{Nat} \rightsquigarrow \mathit{const}\;\mathsf{Nat}$ maps a natural number x to any natural number that is at least x. The conversion isomorphisms (5.2) specialise for BVec to

$$\mathsf{BVec}\; A\; m \;\cong\; \Sigma[\, \mathit{as} : \mathsf{List}\; A\,]\;\; (\![\, \mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg}\,]\!)\; \mathit{as}\; m$$

for all $m \,:\, \mathsf{Nat}$. But is BVec really the bounded vectors? Indeed it is, because we can deduce

$$\mathit{geq} \bullet (\![\, \mathit{fun}\; \mathit{length\text{-}alg}\,]\!) \;\simeq\; (\![\, \mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg}\,]\!)$$

by Fold Fusion. The fusion condition is

$$\mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg} \;\simeq\; \mathit{geq} \bullet \mathit{fun}\; \mathit{length\text{-}alg} \bullet \mathbb{R}\; (\mathit{ListD}\; A)\; \mathit{geq}$$

The left-to-right inclusion is easily calculated as follows:

$$\begin{array}{ll}
& \mathit{geq} \bullet \mathit{fun}\ \mathit{length\text{-}alg} \\
\subseteq & \{\ \text{identity}\ \} \\
& \mathit{geq} \bullet \mathit{fun}\ \mathit{length\text{-}alg} \bullet \mathit{idR} \\
\subseteq & \{\ \text{relator preserves identity}\ \} \\
& \mathit{geq} \bullet \mathit{fun}\ \mathit{length\text{-}alg} \bullet \mathbb{R}\ (\mathit{ListD}\ A)\ \mathit{idR} \\
\subseteq & \{\ \mathit{geq}\ \text{reflexive; relator is monotonic}\ \} \\
& \mathit{geq} \bullet \mathit{fun}\ \mathit{length\text{-}alg} \bullet \mathbb{R}\ (\mathit{ListD}\ A)\ \mathit{geq}
\end{array}$$

And from right to left:

$$\begin{array}{ll}
& \mathit{geq} \bullet \mathit{fun}\ \mathit{length\text{-}alg} \bullet \mathbb{R}\ (\mathit{ListD}\ A)\ \mathit{geq} \\
\subseteq & \{\ \mathit{fun}\ \mathit{length\text{-}alg}\ \text{monotonic on}\ \mathit{geq}\ \} \\
& \mathit{geq} \bullet \mathit{geq} \bullet \mathit{fun}\ \mathit{length\text{-}alg} \\
\subseteq & \{\ \mathit{geq}\ \text{transitive}\ \} \\
& \mathit{geq} \bullet \mathit{fun}\ \mathit{length\text{-}alg}
\end{array}$$

Note that these calculations are good illustrations of the power of relational calculation because of their simplicity — they are straightforward symbol manipulations, hiding details like quantifier reasoning behind the scenes. As demonstrated by the AoPA library [Mu et al., 2009], they can be faithfully formalised with preorder reasoning combinators in AGDA and used to discharge the fusion conditions of $\mathit{fusion\text{-}conversion}_{\subseteq}$ and $\mathit{fusion\text{-}conversion}_{\supseteq}$. Hence we get two conversions, one of type

$$\mathsf{Vec}\ A\ n \rightarrow (n \leqslant m) \rightarrow \mathsf{BVec}\ A\ m$$

which relaxes a vector of length n to a bounded vector whose length is bounded above by some m that is at least n, and the other of type

$$\mathsf{BVec}\ A\ m \rightarrow \Sigma[\,n : \mathsf{Nat}\,]\ \ \mathsf{Vec}\ A\ n \times (n \leqslant m)$$

which converts a bounded vector whose length is at most m to a vector of length precisely n and guarantees that n is at most m. Both conversions preserve the underlying list by construction.

5.3.2 The Streaming Theorem for list metamorphisms

A **metamorphism** [Gibbons, 2007] is an unfold after a fold — it consumes a data structure to compute an intermediate value and then produces a new data structure using the intermediate value as the seed. In this section we will restrict ourselves to metamorphisms consuming and producing lists. As Gibbons noted, (list) metamorphisms in general cannot be automatically optimised in terms of time and space, but under certain conditions it is possible to refine a list metamorphism to a **streaming algorithm** — which can produce an initial segment of the output list without consuming all of the input list — or a parallel algorithm [Nakano, 2013]. In the rest of this section, we prove the **Streaming Theorem** [Bird and Gibbons, 2003, Theorem 30] by implementing the streaming algorithm given by the theorem with algebraically ornamented lists such that the algorithm satisfies its metamorphic specification by construction.

Our first step is to formulate a metamorphism as a relational specification of the streaming algorithm.

- The fold part needs a twist since using the conventional fold — known as the **right fold** for lists since the direction of computation on a list is from right to left (cf. wind direction) — does not easily give rise to a streaming algorithm. This is because we wish to talk about "partial consumption" naturally: for a list, partial consumption means examining and removing some elements of the list to get a sub-list on which we can resume consumption, and the natural way to do this is to consume the list from the left, examining and removing head elements and keeping the tail. We should thus use the **left fold** instead, which is usually defined as

  ```
  foldl : {A X : Set} → (X → A → X) → X → List A → X
  foldl f x []         = x
  foldl f x (a :: as)  = foldl f (f x a) as
  ```

  The connection to the conventional fold (and thus algebraic ornamentation) is not lost, however — it is well known that a left fold can be alternatively implemented as a right fold by turning a list into a chain of functions of type $X \rightarrow X$ transforming the initial value to the final result:

  ```
  foldl-alg : {A X : Set} → (X → A → X) →
  ```

$$\begin{aligned}
&\qquad\qquad \mathbb{F}\ (\mathit{ListD}\ A)\ (\mathit{const}\ (X \to X)) \rightrightarrows \mathit{const}\ (X \to X)\\
&\mathit{foldl\text{-}alg}\ f\ (\text{`nil}\ ,\ \blacksquare) = \mathit{id}\\
&\mathit{foldl\text{-}alg}\ f\ (\text{`cons}\ ,\ a\ ,\ h\ ,\ \blacksquare) = h \circ \mathit{flip}\ f\ a
\end{aligned}$$

$$\begin{aligned}
&\mathit{foldl} : \{A\ X : \mathsf{Set}\} \to (X \to A \to X) \to X \to \mathsf{List}\ A \to X\\
&\mathit{foldl}\ f\ x\ \mathit{as} = \mathit{fold}\ (\mathit{foldl\text{-}alg}\ f)\ \mathit{as}\ x
\end{aligned}$$

The left fold can thus be linked to the relational fold by

$$\mathit{fun}\ (\mathit{foldl}\ f\ x) \simeq \mathit{fun}\ (\lambda\, h \mapsto h\ x) \bullet (\!|\ \mathit{fun}\ (\mathit{foldl\text{-}alg}\ f)\ |\!) \tag{5.5}$$

- The unfold part is approximated by the converse of a relational fold: given a list coalgebra $g : \mathit{const}\ X \rightrightarrows \mathbb{F}\ (\mathit{ListD}\ B)\ (\mathit{const}\ X)$ for some $X : \mathsf{Set}$, we take its converse, turning it into a relational algebra, and use the converse of the relational fold with this algebra.

  $$(\!|\ \mathit{fun}\ g^{\circ}\ |\!)^{\circ} : \mathit{const}\ X \rightsquigarrow \mathit{const}\ (\mathsf{List}\ A)$$

  This is only an approximation because, while the relation does produce a list, the resulting list is inductive rather than coinductive, so the relation is actually a **well-founded** unfold, which is incapable of producing an infinite list.

Thus, given a "left algebra" for consuming List A

$$f : X \to A \to X$$

and a coalgebra for producing List B

$$g : \mathit{const}\ X \rightrightarrows \mathbb{F}\ (\mathit{ListD}\ B)\ (\mathit{const}\ X)$$

which together satisfy a **streaming condition** that we will see later, the streaming algorithm we implement, which takes as input the initial value $x : X$ for the left fold, should be included in the following metamorphic relation:

$$\mathit{meta}\ f\ g\ x = (\!|\ \mathit{fun}\ g^{\circ}\ |\!)^{\circ} \bullet \mathit{fun}\ (\mathit{foldl}\ f\ x) : \mathit{const}\ (\mathsf{List}\ A) \rightsquigarrow \mathit{const}\ (\mathsf{List}\ B)$$

Next we devise a type for the streaming algorithm that fully guarantees its correctness. By (5.5), the specification $\mathit{meta}\ f\ g\ x$ is equivalent to

$$(\!|\ \mathit{fun}\ g^{\circ}\ |\!)^{\circ} \bullet \mathit{fun}\ (\lambda\, h \mapsto h\ x) \bullet (\!|\ \mathit{fun}\ (\mathit{foldl\text{-}alg}\ f)\ |\!)$$

Inspecting the above relation, we see that a conforming program takes a List A that folds to some $h : X \to X$ with $\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f)$ and unfolds a List B from

$h\ x\ :\ X$ with *fun g* (i.e., computes a List B that folds to $h\ x$ with $\mathit{fun}\ g^{\circ}$). We thus implement the streaming algorithm by

$$\begin{aligned} &\mathit{stream}\ f\ g\ :\ (x\ :\ X)\ \{h\ :\ X \to X\} \to \\ &\qquad\qquad \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ h \to \mathsf{AlgList}\ B\ (\mathit{fun}\ g^{\circ})\ (h\ x) \end{aligned}$$

from which we can extract

$$\mathit{stream}'\ f\ g\ :\ X \to \mathsf{List}\ A \to \mathsf{List}\ B$$

which is guaranteed to satisfy

$$\mathit{fun}\ (\mathit{stream}'\ f\ g\ x)\ \subseteq\ \mathit{meta}\ f\ g\ x$$

The extraction of $\mathit{stream}'\ f\ g$ from $\mathit{stream}\ f\ g$ is done with the help of the conversion isomorphisms (5.2) for the two AlgList datatypes involved:

$$\begin{aligned} &\mathit{consumption\text{-}iso}\ : \\ &\quad (h\ :\ X \to X) \to \\ &\quad \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ h\ \cong\ \Sigma[\,\mathit{as} : \mathsf{List}\ A\,]\ \mathit{fold}\ (\mathit{foldl\text{-}alg}\ f)\ \mathit{as}\ \equiv\ h \\ &\mathit{production\text{-}iso}\ : \\ &\quad (x\ :\ X) \to \mathsf{AlgList}\ B\ (\mathit{fun}\ g^{\circ})\ x\ \cong\ \Sigma[\,\mathit{bs} : \mathsf{List}\ B\,]\ (\!|\mathit{fun}\ g^{\circ}|\!)\ \mathit{bs}\ x \end{aligned}$$

(where *consumption-iso* has been simplified by *fun-preserves-fold*). Given $x\ :\ X$, what $\mathit{stream}'\ f\ g\ x$ does is

- lifting the input $\mathit{as}\ :\ \mathsf{List}\ A$ to an algebraically ornamented list of type

  $$\mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ (\mathit{fold}\ (\mathit{foldl\text{-}alg}\ f)\ \mathit{as})$$

  using the right-to-left direction of $\mathit{consumption\text{-}iso}\ (\mathit{fold}\ (\mathit{foldl\text{-}alg}\ f)\ \mathit{as})$ (with the equality proof obligation discharged trivially by refl),
- transforming this algebraically ornamented list to a new one of type

  $$\mathsf{AlgList}\ B\ (\mathit{fun}\ g^{\circ})\ (\mathit{foldl}\ f\ x\ \mathit{as})$$

  using $\mathit{stream}\ f\ g\ x$, and
- demoting the new algebraically ornamented list to List B using the left-to-right direction of $\mathit{production\text{-}iso}\ (\mathit{foldl}\ f\ x\ \mathit{as})$.

The use of *production-iso* in the last step ensures that the result $\mathit{stream}'\ f\ g\ x\ \mathit{as}\ :\ \mathsf{List}\ B$ satisfies

$$(\!|\mathit{fun}\ g^{\circ}|\!)\ (\mathit{stream}'\ f\ g\ x\ \mathit{as})\ (\mathit{foldl}\ f\ x\ \mathit{as})$$

which easily implies

$$((\!(\![\, \mathit{fun}\ g^{\circ}\,]\!)\!)^{\circ} \cdot \mathit{fun}\ (\mathit{foldl}\ f\ x))\ \mathit{as}\ (\mathit{stream}'\ f\ g\ x\ \mathit{as})$$

i.e., $\mathit{fun}\ (\mathit{stream}'\ f\ g\ x) \subseteq \mathit{meta}\ f\ g\ x$, as required.

What is left is the implementation of *stream f g*. Operationally, we maintain a state of type X (and hence require an initial state as an input to the function), and we can try either

- to update the state by consuming elements of A with f, or
- to produce elements of B (and transit to a new state) by applying g to the state.

Since we want *stream f g* to be as productive as possible, we should always try to produce elements of B with g first, and only try to consume elements of A with f when g produces nothing. In AGDA:

```
stream f g : (x : X) {h : X → X} →
             AlgList A (fun (foldl-alg f)) h → AlgList B (fun g °) (h x)
stream f g x     as                    with g x    | inspect g x
stream f g x {h} as                    | next b x' | [ gxeq ] = cons b (h x') { }0
                                                                  (stream f g x' as)
stream f g x     (nil        refl   )  | nothing   | [ gxeq ] = nil gxeq
stream f g x     (cons a h'  refl as)  | nothing   | [ gxeq ] = stream f g (f x a) as
```

We match $g\ x$ with either of the two patterns $\mathsf{next}\ b\ x' = (\text{'}\mathsf{cons}\ , b\ , x'\ , \blacksquare)$ and $\mathsf{nothing} = (\text{'}\mathsf{nil}\ , \blacksquare)$.

- If the result is $\mathsf{next}\ b\ x'$, we should emit b and use x' as the new state; the recursively computed algebraically ornamented list is indexed with $h\ x'$, and we are left with a proof obligation of type $g\ (h\ x) \equiv \mathsf{next}\ b\ (h\ x')$ at Goal 0; we will come back to this proof obligation later.
- If the result is nothing, we should attempt to consume from the input list.
  - If the input list is empty, implying that the index h of its type is just *id*, both production and consumption have ended, so we return an empty list. The nil constructor requires a proof of $(\mathit{fun}\ g^{\circ})\ \mathsf{nothing}\ (h\ x)$, which reduces to $g\ x \equiv \mathsf{nothing}$ and is discharged with the help of the "inspect idiom" in AGDA's standard library (which, in a **with**-matching, gives a proof that

$$\begin{tikzcd} x \arrow[r, mapsto, "\text{produce } b \text{ with } g"] \arrow[d, mapsto, "\text{consume } \mathit{as} \text{ with } h"'] & x' \arrow[d, mapsto, "\text{consume } \mathit{as} \text{ with } h"] \\ h\;x \arrow[r, mapsto, "\text{produce } b \text{ with } g"'] & h\;x' \end{tikzcd}$$

Figure 5.5 State transitions involved in commutativity of production and consumption (cf. Gibbons [2007, Figures 1 and 2]).

the term being matched (in this case $g\ x$) is propositionally equal to the matched pattern (in this case $\mathsf{nothing}$)).

– Otherwise the input list is nonempty, implying that h is $h' \circ \mathit{flip}\ f\ a$ where a is the head of the input list, and we should continue with the new state $f\ x\ a$, keeping the tail for further consumption. Typing directly works out because the index of the recursive result $h'\ (f\ x\ a)$ and the required index $(h' \circ \mathit{flip}\ f\ a)\ x$ are definitionally equal.

Now we look at Goal 0. We have

$$\mathit{gxeq}\ :\ g\ x\ \equiv\ \mathsf{next}\ b\ x'$$

in the context, and need to prove

$$g\ (h\ x)\ \equiv\ \mathsf{next}\ b\ (h\ x')$$

This is commutativity of production and consumption (see Figure 5.5): The function $h\ :\ X \to X$ is the state transformation resulting from consumption of the input list *as*. From the initial state x, we can either

- apply g to x to produce b and reach a new state x', and then apply h to consume the list and update the state to $h\ x'$, or
- apply h to consume the list and update the state to $h\ x$, and then apply g to $h\ x$ to produce an element and reach a new state,

and we need to prove that the outcomes are the same: doing production using g and consumption using h in whichever order should emit the same element and reach the same final state. This cannot be true in general, so we should impose some commutativity condition on f and g, which is called the **streaming condition**:

$$\begin{array}{l} \mathit{StreamingCondition}\ f\ g \ :\ \mathsf{Set} \\ \mathit{StreamingCondition}\ f\ g \ = \\ \quad (a\ :\ A)\ (b\ :\ B)\ (x\ x'\ :\ X) \to g\ x\ \equiv\ \mathsf{next}\ b\ x' \to g\ (f\ x\ a)\ \equiv\ \mathsf{next}\ b\ (f\ x'\ a) \end{array}$$

The streaming condition is commutativity of one step of production and consumption, whereas the proof obligation at Goal 0 is commutativity of one step of production and multiple steps of consumption (of the entire list), so we perform a straightforward induction to extend the streaming condition along the axis of consumption:

$$\begin{array}{l} \mathit{streaming\text{-}lemma}\ : \\ \quad (b\ :\ B)\ (x\ x'\ :\ X) \to g\ x\ \equiv\ \mathsf{next}\ b\ x' \to \\ \quad \{h\ :\ X \to X\} \to \mathsf{AlgList}\ A\ (\mathit{fun}\ (\mathit{foldl\text{-}alg}\ f))\ h \to g\ (h\ x)\ \equiv\ \mathsf{next}\ b\ (h\ x') \\ \mathit{streaming\text{-}lemma}\ b\ x\ x'\ \mathit{eq}\ (\mathsf{nil}\qquad \mathsf{refl}\quad)\ =\ \mathit{eq} \\ \mathit{streaming\text{-}lemma}\ b\ x\ x'\ \mathit{eq}\ (\mathsf{cons}\ a\ h\ \mathsf{refl}\ \mathit{as})\ = \\ \quad \mathit{streaming\text{-}lemma}\ b\ (f\ x\ a)\ (f\ x'\ a)\ (\mathit{streaming\text{-}condition}\ f\ g\ a\ b\ x\ x'\ \mathit{eq})\ \mathit{as} \end{array}$$

where *streaming-condition* : *StreamingCondition f g* is a proof term that should be supplied along with f and g in the beginning. Goal 0 is then discharged by the term *streaming-lemma b x x′ gxeq as*.

We have thus completed the implementation of the Streaming Theorem, except that *stream f g* is non-terminating, as there is no guarantee that g produces only a finite number of elements. In our setting, where the output list is specified to be finite, we can additionally require that g is well-founded and revise *stream* accordingly (see, e.g., Nordström [1988]); the general way out is to switch to coinductive datatypes to allow the output list to be infinite, which, however, falls outside the scope of this dissertation.

It is interesting to compare our implementation with the proofs of Bird and Gibbons [2003]. While their Lemma 29 turns explicitly into our *streaming-lemma*, their Theorem 30 goes implicitly into the typing of *stream* and no longer needs special attention. The structure of *stream* already matches that of Bird and Gibbons's proof of their Theorem 30, and the principled type design using algebraic ornamentation elegantly loads the proof onto the structure of *stream* —

this is internalism at its best.

5.3.3 The Greedy Theorem and the minimum coin change problem

Suppose that we have an unlimited number of 1-penny, 2-pence, and 5-pence coins, modelled by the following datatype:

```
data Coin : Set where
  1p : Coin
  2p : Coin
  5p : Coin
```

The **minimum coin change problem** asks for the least number of coins that make up n pence for any given n : Nat. We can give a relational specification of the problem with the following **minimisation** operator:

$$\begin{aligned}&\mathit{min}\_\bullet\Lambda\_ : \{I : \mathsf{Set}\}\,\{X\,Y : I \to \mathsf{Set}\}\,(R : Y \rightsquigarrow Y)\,(S : X \rightsquigarrow Y) \to (X \rightsquigarrow Y)\\&\mathit{min}\_\bullet\Lambda\_\ \{Y := Y\}\ R\ S\ \{i\}\ x\ y \ = \ S\ x\ y \times ((y' : Y\ i) \to S\ x\ y' \to R\ y'\ y)\end{aligned}$$

An input $x : X\ i$ for some $i : I$ is mapped by $\mathit{min}\ R\ \bullet\Lambda\ S$ to $y : Y\ i$ if y is a possible result in $S\ x : \mathscr{P}(Y\ i)$ and is the smallest such result under R, in the sense that any y' in $S\ x : \mathscr{P}(Y\ i)$ must satisfy $R\ y'\ y$. (We think of R as mapping larger inputs to smaller outputs.) Intuitively, we can think of $\mathit{min}\ R\ \bullet\Lambda\ S$ as consisting of two steps: the first step $\Lambda\ S$ computes the set of all possible results yielded by S, and the second step $\mathit{min}\ R$ nondeterministically chooses a minimum result from that set. We use bags of coins as the type of solutions, and represent them as decreasingly ordered lists indexed with an upper bound. (This is a deliberate choice to make the derivation work, but one would naturally be led to this representation having attempted to apply the **Greedy Theorem**, which will be introduced shortly.) If we define the ordering on coins as

$$\begin{aligned}&\_\leqslant_{\mathsf{C}}\_ : \mathsf{Coin} \to \mathsf{Coin} \to \mathsf{Set}\\&c \leqslant_{\mathsf{C}} d \ = \ \mathit{value}\ c \leqslant \mathit{value}\ d\end{aligned}$$

where the values of the coins are defined by

$$\begin{aligned}
&\mathit{value} : \mathsf{Coin} \to \mathsf{Nat}\\
&\mathit{value}\ \mathsf{1p} = 1\\
&\mathit{value}\ \mathsf{2p} = 2\\
&\mathit{value}\ \mathsf{5p} = 5
\end{aligned}$$

then the datatype of coin bags we use is

$\mathit{CoinBagOD} : \mathsf{OrnDesc}\ \mathsf{Coin}\ !\ (\mathit{ListD}\ \mathsf{Coin})$
$\mathit{CoinBagOD} = \mathit{OrdListOD}\ \mathsf{Coin}\ (\mathit{flip}\ \_{\leqslant_{\mathsf{C}}}\_)$
  -- specialising *Val* to Coin and $\_\leqslant\_$ to *flip* $\_{\leqslant_{\mathsf{C}}}\_$

indexfirst data $\mathsf{CoinBag} : \mathsf{Coin} \to \mathsf{Set}$ **where**
  $\mathsf{CoinBag}\ c \ni \mathsf{nil}$
    $\mid \mathsf{cons}\ (d : \mathsf{Coin})\ (\mathit{leq} : d \leqslant_{\mathsf{C}} c)\ (b : \mathsf{CoinBag}\ d)$

The total value of a coin bag is the sum of the values of the coins in the bag, which is computed by a (functional) fold:

$\mathit{total\text{-}value\text{-}alg} : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat}$
$\mathit{total\text{-}value\text{-}alg}\ (\text{‘}\mathsf{nil}\ ,\ \blacksquare) = 0$
$\mathit{total\text{-}value\text{-}alg}\ (\text{‘}\mathsf{cons}\ , d\ , \_\ , n\ , \blacksquare) = \mathit{value}\ d + n$

$\mathit{total\text{-}value} : \mathsf{CoinBag} \rightrightarrows \mathit{const}\ \mathsf{Nat}$
$\mathit{total\text{-}value} = \mathit{fold}\ \mathit{total\text{-}value\text{-}alg}$

and the number of coins in a coin bag is computed by an ornamental forgetful function shrinking ordered lists to natural numbers:

$\mathit{size\text{-}alg} : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat}$
$\mathit{size\text{-}alg} = \mathit{ornAlg}\ (\mathit{NatD\text{-}ListD}\ \mathsf{Coin} \odot \lceil \mathit{CoinBagOD} \rceil)$

$\mathit{size} : \mathsf{CoinBag} \rightrightarrows \mathit{const}\ \mathsf{Nat}$
$\mathit{size} = \mathit{fold}\ \mathit{size\text{-}alg}$
  -- which is definitionally *forget* $(\mathit{NatD\text{-}ListD}\ \mathsf{Coin} \odot \lceil \mathit{CoinBagOD} \rceil)$

The specification of the minimum coin change problem can now be written as

$\mathit{min\text{-}coin\text{-}change} : \mathit{const}\ \mathsf{Nat} \leadsto \mathsf{CoinBag}$
$\mathit{min\text{-}coin\text{-}change} = \mathit{min}\ (\mathit{fun}\ \mathit{size}^{\circ} \bullet \mathit{leq} \bullet \mathit{fun}\ \mathit{size}) \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value}^{\circ})$

Intuitively, given an input $n : \mathsf{Nat}$, the relation $\mathit{fun}\ \mathit{total\text{-}value}^{\circ}$ computes an arbitrary coin bag whose total value is n, so *min-coin-change* first computes the set of all such coin bags and then chooses from the set a coin bag whose size is smallest. Our goal, then, is to write a functional program $f : \mathit{const}\ \mathsf{Nat} \rightrightarrows \mathsf{CoinBag}$ such that $\mathit{fun}\ f \subseteq \mathit{min\text{-}coin\text{-}change}$, and then $f\ \{\mathsf{5p}\} : \mathsf{Nat} \to \mathsf{CoinBag}\ \mathsf{5p}$ would be a solution. (The type $\mathsf{CoinBag}\ \mathsf{5p}$ contains all coin bags, since $\mathsf{5p}$ is the largest denomination and hence a trivial upper bound on the content of bags.) Of course, we may guess what f should look like, but its correctness proof is much harder. Can we construct the program and its correctness proof in a more manageable way?

The plan

In traditional relational program derivation [Bird and de Moor, 1997], we would attempt to refine the specification *min-coin-change* to some simpler relational program and then to an executable functional program by applying algebraic laws and theorems. With algebraic ornamentation, however, there is a new possibility: if, for some algebra $R : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat}) \rightsquigarrow \mathit{const}\ \mathsf{Nat}$, we can derive

$$(\!|\, R \,|\!)^{\circ} \subseteq \mathit{min\text{-}coin\text{-}change} \tag{5.6}$$

then we can manufacture a new datatype

$$\begin{aligned}
&\mathit{GreedyBagOD} : \mathsf{OrnDesc}\ (\mathsf{Coin} \times \mathsf{Nat})\ \mathit{outl}\ \lfloor \mathit{CoinBagOD} \rfloor \\
&\mathit{GreedyBagOD} = \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ R
\end{aligned}$$

$$\begin{aligned}
&\mathsf{GreedyBag} : \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Set} \\
&\mathsf{GreedyBag}\ c\ n = \mu\ \lfloor \mathit{GreedyBagOD} \rfloor\ (c\ ,\ n)
\end{aligned}$$

and construct a function of type

$$\mathit{greedy} : (c : \mathsf{Coin})\ (n : \mathsf{Nat}) \to \mathsf{GreedyBag}\ c\ n$$

from which we can assemble a solution

$$\begin{aligned}
&\mathit{sol} : \mathsf{Nat} \to \mathsf{CoinBag}\ \mathsf{5p} \\
&\mathit{sol} = \mathit{forget}\ \lceil \mathit{GreedyBagOD} \rceil \circ \mathit{greedy}\ \mathsf{5p}
\end{aligned}$$

The program *sol* satisfies the specification because of the following argument: For any $c : \mathsf{Coin}$ and $n : \mathsf{Nat}$, by (5.2) we have

$$\mathsf{GreedyBag}\ c\ n \cong \Sigma[\, b : \mathsf{CoinBag}\ c \,]\ (\!|\, R \,|\!)\ b\ n$$

In particular, since the first half of the left-to-right direction of the isomorphism is *forget* $\lceil$*GreedyBagOD*$\rceil$, we have

$$(\![\, R \,]\!)\ (\mathit{forget}\ \lceil \mathit{GreedyBagOD} \rceil\ g)\ n$$

for any $g\ :\ \mathsf{GreedyBag}\ c\ n$. Substituting g by *greedy* $\mathsf{5p}\ n$, we get

$$(\![\, R \,]\!)\ (\mathit{sol}\ n)\ n$$

which implies, by (5.6),

$$\mathit{min\text{-}coin\text{-}change}\ n\ (\mathit{sol}\ n)$$

i.e., *sol* satisfies the specification. Thus all we need to do to solve the minimum coin change problem is

- refine the specification *min-coin-change* to the converse of a fold, i.e., find the algebra R in (5.6), and
- construct the internalist program *greedy*.

Refining the specification

The key to refining *min-coin-change* to the converse of a fold lies in the following version of the **Greedy Theorem**, which is a specialisation of Bird and de Moor's Theorem 10.1 modulo indexing: Let $D\ :\ \mathsf{Desc}\ I$ be a description, $R\ :\ \mu\ D \rightsquigarrow \mu\ D$ a preorder, and $S\ :\ \mathbb{F}\ D\ X \rightsquigarrow X$ an algebra. Consider the specification

$$\mathit{min}\ R \bullet \Lambda\ ((\![\, S \,]\!)^{\circ})$$

That is, given an input value $x\ :\ X\ i$ for some $i\ :\ I$, we choose a minimum under R among all those elements of $\mu\ D\ i$ that computes to x through $(\![\, S \,]\!)$. The Greedy Theorem states that, if the initial algebra

$$\alpha\ =\ \mathit{fun}\ \mathsf{con}\ :\ \mathbb{F}\ D\ (\mu\ D) \rightsquigarrow \mu\ D$$

is monotonic on R, i.e.,

$$\alpha \bullet \mathbb{R}\ D\ R \subseteq R \bullet \alpha$$

and there is a relation (ordering) $Q\ :\ \mathbb{F}\ D\ X \rightsquigarrow \mathbb{F}\ D\ X$ such that the **greedy condition**

$$\alpha \bullet \mathbb{R}\ D\ ((\![\, S \,]\!)^{\circ}) \bullet (Q \cap (S^{\circ} \bullet S))^{\circ} \subseteq R^{\circ} \bullet \alpha \bullet \mathbb{R}\ D\ ((\![\, S \,]\!)^{\circ})$$

is satisfied, then we have

$$(\!|\, (\mathit{min}\ Q \bullet \Lambda\ (S\,^\circ))\,^\circ\, |\!)^\circ \subseteq \mathit{min}\ R \bullet \Lambda\ ((\!|\, S\, |\!)^\circ)$$

The Greedy Theorem essentially reduces a <u>global</u> optimisation problem (as indicated by the <u>outermost</u> *min R*) to a <u>local</u> optimisation problem (as indicated by the *min Q* <u>inside</u> the relational fold). The theorem admits an elegant calculational proof, which can be faithfully reprised in AGDA. Here we offer an intuitive explanation: The monotonicity condition states that if $ds : \mathbb{F}\ D\ (\mu\ D)\ i$ for some $i : I$ is better than $ds' : \mathbb{F}\ D\ (\mu\ D)\ i$ under $\mathbb{R}\ D\ R$, i.e., ds and ds' are equal except that the recursive positions of ds are all better than the corresponding recursive positions of ds' under R, then $\mathsf{con}\ ds : \mu\ D\ i$ would be better than $\mathsf{con}\ ds' : \mu\ D\ i$ under R. This implies that, when solving the optimisation problem, better solutions to sub-problems would lead to a better solution to the original problem, so the **principle of optimality** applies — to reach an optimal solution, it suffices to find optimal solutions to sub-problems, and we are entitled to use the converse of a fold to find optimal solutions recursively. The greedy condition further states that there is an ordering Q on the ways of decomposing the problem that has significant influence on the quality of solutions: Suppose that there are two decompositions xs and $xs' : \mathbb{F}\ D\ X\ i$ of some problem $x : X\ i$ for some $i : I$, i.e., both xs and xs' are in $(S\,^\circ)\ x : \mathscr{P}(\mathbb{F}\ D\ X\ i)$, and assume that xs is better than xs' under Q. Then for any solution resulting from xs' (computed by $\alpha \bullet \mathbb{R}\ D\ ((\!|\, S\, |\!)^\circ))$ there always exists a better solution resulting from xs, so ignoring xs' would only rule out worse solutions. The greedy condition thus guarantees that we will arrive at an optimal solution by always choosing the best decomposition, which is done by $\mathit{min}\ Q \bullet \Lambda\ (S\,^\circ) : X \rightsquigarrow \mathbb{F}\ D\ X$.

Back to the minimum coin change problem. By *fun-preserves-fold*, the specification *min-coin-change* is equivalent to

$$\mathit{min}\ (\mathit{fun\ size}^\circ \bullet \mathit{leq} \bullet \mathit{fun\ size}) \bullet \Lambda\ ((\!|\, \mathit{fun\ total\text{-}value\text{-}alg}\, |\!)^\circ)$$

which matches the form of the generic specification given in the Greedy Theorem, so we try to discharge the two conditions of the theorem. The monotonicity condition reduces to monotonicity of *fun size-alg* on *leq*, and can be easily proved either by relational calculation or pointwise reasoning. As for the greedy condition, an obvious choice for Q is an ordering that leads us to choose the largest

```
data CoinOrderedView : Coin → Coin → Set where
  1p1p  : CoinOrderedView 1p 1p
  1p2p  : CoinOrderedView 1p 2p
  1p5p  : CoinOrderedView 1p 5p
  2p2p  : CoinOrderedView 2p 2p
  2p5p  : CoinOrderedView 2p 5p
  5p5p  : CoinOrderedView 5p 5p
view-ordered-coin : (c d : Coin) → c ⩽C d → CoinOrderedView c d
data CoinBag′View : {c : Coin} {n : Nat} {l : Nat} → CoinBag′ c n l → Set where
  empty : {c : Coin} → CoinBag′View {c} {0} {0} bnil
  1p1p  : {m l : Nat} {lep : 1p ⩽C 1p}
          (b : CoinBag′ 1p m l) → CoinBag′View {1p} {1 + m} {1 + l} (bcons 1p lep b)
  1p2p  : {m l : Nat} {lep : 1p ⩽C 2p}
          (b : CoinBag′ 1p m l) → CoinBag′View {2p} {1 + m} {1 + l} (bcons 1p lep b)
  2p2p  : {m l : Nat} {lep : 2p ⩽C 2p}
          (b : CoinBag′ 2p m l) → CoinBag′View {2p} {2 + m} {1 + l} (bcons 2p lep b)
  1p5p  : {m l : Nat} {lep : 1p ⩽C 5p}
          (b : CoinBag′ 1p m l) → CoinBag′View {5p} {1 + m} {1 + l} (bcons 1p lep b)
  2p5p  : {m l : Nat} {lep : 2p ⩽C 5p}
          (b : CoinBag′ 2p m l) → CoinBag′View {5p} {2 + m} {1 + l} (bcons 2p lep b)
  5p5p  : {m l : Nat} {lep : 5p ⩽C 5p}
          (b : CoinBag′ 5p m l) → CoinBag′View {5p} {5 + m} {1 + l} (bcons 5p lep b)
view-CoinBag′ : {c : Coin} {n l : Nat} (b : CoinBag′ c n l) → CoinBag′View b
```

Figure 5.6 Two views for proving *greedy-lemma*.

```
greedy-lemma : (c d : Coin) → c ≤_C d → (m n : Nat) → value c + m ≡ value d + n →
               (l : Nat) (b : CoinBag′ c m l) → Σ[ l′ : Nat ] CoinBag′ d n l′ × (l′ ≤ l)
greedy-lemma c   d   c≤d m       n   eq  l            b with view-ordered-coin c d c≤d
greedy-lemma .1p .1p _   .n      n   refl l           b^{CoinBag′ 1p n l} | 1p1p = { Σ[ l′ : Nat ] CoinBag′ 1p n l′ × (l′ ≤ l) }_0
greedy-lemma .1p .2p _   .(1 + n) n  refl l           b | 1p2p with view-CoinBag′ b
greedy-lemma .1p .2p _   .(1 + n) n  refl .(1 + l″) ._ | 1p2p | 1p1p {.n} {l″} b^{CoinBag′ 1p n l″} =
                                                          { Σ[ l′ : Nat ] CoinBag′ 2p n l′ × (l′ ≤ 1 + l″) }_1
greedy-lemma .1p .5p _   .(4 + n) n  refl l           b | 1p5p with view-CoinBag′ b
greedy-lemma .1p .5p _   .(4 + n) n  refl ._         ._ | 1p5p | 1p1p b with view-CoinBag′ b
greedy-lemma .1p .5p _   .(4 + n) n  refl ._         ._ | 1p5p | 1p1p ._ | 1p1p b with view-CoinBag′ b
greedy-lemma .1p .5p _   .(4 + n) n  refl ._         ._ | 1p5p | 1p1p ._ | 1p1p ._ | 1p1p b with view-CoinBag′ b
greedy-lemma .1p .5p _   .(4 + n) n  refl .(4 + l″) ._ | 1p5p | 1p1p ._ | 1p1p ._ | 1p1p ._ | 1p1p {.n} {l″} b^{CoinBag′ 1p n l″} =
                                                          { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ≤ 4 + l″) }_2
greedy-lemma .2p .2p _   .n      n   refl l           b^{CoinBag′ 2p n l} | 2p2p = { Σ[ l′ : Nat ] CoinBag′ 2p n l′ × (l′ ≤ l) }_3
greedy-lemma .2p .5p _   .(3 + n) n  refl l           b | 2p5p with view-CoinBag′ b
greedy-lemma .2p .5p _   .(3 + n) n  refl ._         ._ | 2p5p | 1p2p b with view-CoinBag′ b
greedy-lemma .2p .5p _   .(3 + n) n  refl ._         ._ | 2p5p | 1p2p ._ | 1p1p b with view-CoinBag′ b
greedy-lemma .2p .5p _   .(3 + n) n  refl .(3 + l″) ._ | 2p5p | 1p2p ._ | 1p1p ._ | 1p1p {.n} {l″} b^{CoinBag′ 1p n l″} =
                                                          { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ≤ 3 + l″) }_4
greedy-lemma .2p .5p _   .(3 + n) n  refl ._         ._ | 2p5p | 2p2p b with view-CoinBag′ b
greedy-lemma .2p .5p _   .(3 + n) n  refl .(2 + l″) ._ | 2p5p | 2p2p ._ | 1p2p {.n} {l″} b^{CoinBag′ 2p n l″} =
                                                          { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ≤ 2 + l″) }_5
greedy-lemma .2p .5p _   .(4 + k) .(1 + k) refl .(2 + l″) ._ | 2p5p | 2p2p ._ | 2p2p {k} {l″} b^{CoinBag′ 2p k l″} =
                                                       { Σ[ l′ : Nat ] CoinBag′ 5p (1 + k) l′ × (l′ ≤ 2 + l″) }_6
greedy-lemma .5p .5p _   .n      n   refl l           b^{CoinBag′ 5p n l} | 5p5p = { Σ[ l′ : Nat ] CoinBag′ 5p n l′ × (l′ ≤ l) }_7
```

Figure 5.7 Cases of *greedy-lemma*, generated semi-automatically by AGDA's interactive case-split mechanism. Goal types are shown in the interaction points, and the types of some pattern variables are shown in subscript beside them.

possible denomination, so we go for

$$\begin{aligned}
&Q : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat}) \rightsquigarrow \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat})\\
&Q\ (\text{‘}\mathsf{nil}\quad ,\qquad \blacksquare\,) \;=\; \mathit{return}\ (\text{‘}\mathsf{nil}\ ,\ \blacksquare)\\
&Q\ (\text{‘}\mathsf{cons}\ ,\ d\ ,\ \_) \;=\; (\lambda\, e\ \mathit{rest} \mapsto \text{‘}\mathsf{cons}\ ,\ e\ ,\ \mathit{rest})\ \langle\$\rangle^2\ (\_ \leqslant_{\mathsf{C}} \_\ d)\ \mathit{any}
\end{aligned}$$

where, in the cons case, the output is required to be also a cons node, and the coin at its head position must be one that is no smaller than the coin d at the head position of the input. It is nontrivial to prove the greedy condition by relational calculation. Here we offer instead a brute-force yet conveniently expressed case analysis by pattern matching. Define a new datatype $\mathsf{CoinBag}'$ by composing two algebraic ornaments on $\lfloor \mathit{CoinBagOD} \rfloor$ in parallel:

$$\begin{aligned}
&\mathit{CoinBag'OD} : \mathsf{OrnDesc}\ (\mathit{outl} \bowtie \mathit{outl})\ \mathit{pull}\ \lfloor \mathit{CoinBagOD} \rfloor\\
&\mathit{CoinBag'OD} \;=\; \lceil \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{fun}\ \mathit{total\text{-}value\text{-}alg}) \rceil \otimes\\
&\qquad\qquad\qquad\;\; \lceil \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{fun}\ \mathit{size\text{-}alg}) \rceil
\end{aligned}$$

$$\begin{aligned}
&\mathsf{CoinBag}' : \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}\\
&\mathsf{CoinBag}'\ c\ n\ l \;=\; \mu\ \lfloor \mathit{CoinBag'OD} \rfloor\ (\mathsf{ok}\ (c\ ,\ n)\ ,\ \mathsf{ok}\ (c\ ,\ l))
\end{aligned}$$

whose two constructors can be specialised to

$$\begin{aligned}
&\mathsf{bnil}\quad : \{c : \mathsf{Coin}\} \to \mathsf{CoinBag}'\ c\ 0\ 0\\
&\mathsf{bcons} : \{c : \mathsf{Coin}\}\ \{n\ l : \mathsf{Nat}\} \to (d : \mathsf{Coin}) \to d \leqslant_{\mathsf{C}} c \to\\
&\qquad\quad \mathsf{CoinBag}'\ d\ n\ l \to \mathsf{CoinBag}'\ c\ (\mathit{value}\ d + n)\ (1 + l)
\end{aligned}$$

By predicate swapping using the modularity isomorphisms (**??**) and *fun-preserves-fold*, $\mathsf{CoinBag}'$ is characterised by the isomorphisms

$$\mathsf{CoinBag}'\ c\ n\ l \cong \Sigma[\,b : \mathsf{CoinBag}\ c\,]\ \ (\mathit{total\text{-}value}\ b \equiv n) \times (\mathit{size}\ b \equiv l) \qquad (5.7)$$

for all $c : \mathsf{Coin}$, $n : \mathsf{Nat}$, and $l : \mathsf{Nat}$. Hence a coin bag of type $\mathsf{CoinBag}'\ c\ n\ l$ contains l coins that are no larger than c and sum up to n pence. The greedy condition then essentially reduces to this lemma:

$$\begin{aligned}
\mathit{greedy\text{-}lemma} : {} & (c\ d : \mathsf{Coin}) \to c \leqslant_{\mathsf{C}} d \to\\
& (m\ n : \mathsf{Nat}) \to \mathit{value}\ c + m \equiv \mathit{value}\ d + n \to\\
& (l : \mathsf{Nat})\ (b : \mathsf{CoinBag}'\ c\ m\ l) \to\\
& \Sigma[\,l' : \mathsf{Nat}\,]\ \mathsf{CoinBag}'\ d\ n\ l' \times (l' \leqslant l)
\end{aligned}$$

That is, given a problem (i.e., a value to be represented by coins), if $c : \mathsf{Coin}$ is a choice of decomposition (i.e., the first coin used) no better than $d : \mathsf{Coin}$ (i.e.,

$c \leqslant_{\mathsf{C}} d$ — recall that we prefer larger denominations), and $b : \mathsf{CoinBag}'\ c\ m\ l$ is a solution of size l to the remaining sub-problem m resulting from choosing c, then there is a solution to the remaining sub-problem n resulting from choosing d whose size l' is no greater than l. We define two views (**??**) to aid the analysis, whose datatypes and covering functions are shown in Figure 5.6:

- the first view analyses a proof of $c \leqslant_{\mathsf{C}} d$ and exhausts all possibilities of c and d, and
- the second view analyses some $b : \mathsf{CoinBag}'\ c\ n\ l$ and exhausts all possibilities of c, n, l, and the first coin in b (if any).

The function *greedy-lemma* can then be split into eight cases by first exhausting all possibilities of c and d by the first view and then analysing the content of b by the second view. Figure 5.7 shows the case-split tree generated semi-automatically by AGDA; the detail is explained as follows:

- At Goal 0 (and similarly Goals 3 and 7), the input bag is $b : \mathsf{CoinBag}'\ \mathsf{1p}\ n\ l$, and we should produce a $\mathsf{CoinBag}'\ \mathsf{1p}\ n\ l'$ for some $l' : \mathsf{Nat}$ such that $l' \leqslant l$. This is easy because b itself is a suitable bag.
- At Goal 1 (and similarly Goals 2, 4, and 5), the input bag has type $\mathsf{CoinBag}'\ \mathsf{1p}\ (1 + n)\ l$, i.e., the coins in the bag are no larger than 1p and the total value is $1 + n$. The bag must contain 1p as its first coin; let the rest of the bag be $b : \mathsf{CoinBag}'\ \mathsf{1p}\ n\ l''$. At this point AGDA can deduce that l must be $1 + l''$. Now we can return b as the result after the upper bound on its coins is relaxed from 1p to 2p, which is done by

  cb′-relax :
  $\quad \{c\ d : \mathsf{Coin}\}\ \{n\ l : \mathsf{Nat}\} \rightarrow c \leqslant_{\mathsf{C}} d \rightarrow \mathsf{CoinBag}'\ c\ n\ l \rightarrow \mathsf{CoinBag}'\ d\ n\ l$

- The remaining Goal 6 is the most interesting one: The input bag has type $\mathsf{CoinBag}'\ \mathsf{2p}\ (3 + n)\ l$, which in this case contains two 2-pence coins, and the rest of the bag is $b : \mathsf{CoinBag}'\ \mathsf{2p}\ k\ l''$. AGDA deduces that n must be $1 + k$ and l must be $2 + l''$. We thus need to add a penny to b to increase its total value to $1 + k$, which is done by

  add-penny :
  $\quad \{c : \mathsf{Coin}\}\ \{n\ l : \mathsf{Nat}\} \rightarrow \mathsf{CoinBag}'\ c\ n\ l \rightarrow \mathsf{CoinBag}'\ c\ (1 + n)\ (1 + l)$

and relax the bound of *add-penny b* from 2p to 5p.

The above case analysis may look tedious, but AGDA is able to

- produce all the cases (modulo some cosmetic revisions) after the programmer decides to use the two views and instructs AGDA to do case splitting accordingly, and
- manage all the bookkeeping and deductions about the total value and the size of bags with dependent pattern matching,

so the overhead on the programmer's side is actually less than it seems. The greedy condition can now be discharged by pointwise reasoning, using (5.7) to interface with *greedy-lemma*. We conclude that the Greedy Theorem is applicable, and obtain

$$(\!|\, (\mathit{min}\ Q \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value\text{-}alg}\,^{\circ}))\,^{\circ}\, |\!)\,^{\circ} \subseteq \mathit{min\text{-}coin\text{-}change}$$

We have thus found the algebra

$$R \;=\; (\mathit{min}\ Q \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value\text{-}alg}\,^{\circ}))\,^{\circ}$$

which will help us to construct the final internalist program.

Constructing the internalist program

As planned, we synthesise a new datatype by ornamenting CoinBag using the algebra R derived above:

$$\begin{aligned}
&\mathit{GreedyBagOD} \;:\; \mathsf{OrnDesc}\ (\mathsf{Coin} \times \mathsf{Nat})\ \mathit{outl}\ \lfloor \mathit{CoinBagOD} \rfloor \\
&\mathit{GreedyBagOD} \;=\; \mathit{algOD}\ \lfloor \mathit{CoinBagOD} \rfloor\ R
\end{aligned}$$

$$\begin{aligned}
&\mathsf{GreedyBag} \;:\; \mathsf{Coin} \to \mathsf{Nat} \to \mathsf{Set} \\
&\mathsf{GreedyBag}\ c\ n \;=\; \mu\ \lfloor \mathit{GreedyBagOD} \rfloor\ (c\ ,n)
\end{aligned}$$

whose two constructors can be given the following types:

$$\begin{aligned}
\mathsf{gnil}\ \;:\;& \{c : \mathsf{Coin}\}\ \{n : \mathsf{Nat}\} \to \\
& \mathit{total\text{-}value\text{-}alg}\ (\text{‘}\mathsf{nil}\ , \blacksquare) \equiv n \to \\
& ((\mathit{ns} : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat})) \to \\
& \quad \mathit{total\text{-}value\text{-}alg}\ \mathit{ns} \equiv n \to Q\ \mathit{ns}\ (\text{‘}\mathsf{nil}\ , \blacksquare)) \to \\
& \mathsf{GreedyBag}\ c\ n
\end{aligned}$$

$$\begin{aligned}
&\mathsf{gcons} : \{c : \mathsf{Coin}\}\ \{n : \mathsf{Nat}\}\ (d : \mathsf{Coin})\ (d{\leqslant}c : d \leqslant_{\mathsf{C}} c) \to \\
&\quad \{n' : \mathsf{Nat}\} \to \mathit{total\text{-}value\text{-}alg}\ (\text{`}\mathsf{cons}\ , d\ , d{\leqslant}c\ , n'\ , \blacksquare) \equiv n \to \\
&\quad ((\mathit{ns} : \mathbb{F}\ \lfloor \mathit{CoinBagOD} \rfloor\ (\mathit{const}\ \mathsf{Nat})) \to \\
&\quad\quad \mathit{total\text{-}value\text{-}alg}\ \mathit{ns} \equiv n \to Q\ \mathit{ns}\ (\text{`}\mathsf{cons}\ , d\ , d{\leqslant}c\ , n'\ , \blacksquare)) \to \\
&\quad \mathsf{GreedyBag}\ d\ n' \to \mathsf{GreedyBag}\ c\ n
\end{aligned}$$

and implement the greedy algorithm by

$$\mathit{greedy} : (c : \mathsf{Coin})\ (n : \mathsf{Nat}) \to \mathsf{GreedyBag}\ c\ n$$

Let us first simplify the two constructors of GreedyBag. Each of the two constructors has two additional proof obligations coming from the algebra R:

- For gnil,
  - the first obligation $\mathit{total\text{-}value\text{-}alg}\ (\text{`}\mathsf{nil}\ , \blacksquare) \equiv n$ reduces to $0 \equiv n$, so we may discharge the obligation by specialising n to 0;
  - for the second obligation, ns is necessarily $(\text{`}\mathsf{nil}\ , \blacksquare)$ if $\mathit{total\text{-}value\text{-}alg}\ \mathit{ns} \equiv 0$, and indeed $Q$ maps $(\text{`}\mathsf{nil}\ , \blacksquare)$ to $(\text{`}\mathsf{nil}\ , \blacksquare)$, so the second obligation can be discharged as well.

  We thus obtain a simplified version of gnil:

  $$\mathsf{gnil}' : \{c : \mathsf{Coin}\} \to \mathsf{GreedyBag}\ c\ 0$$

- For gcons,
  - the first obligation reduces to $\mathit{value}\ d + n' \equiv n$, so we may just specialise n to $\mathit{value}\ d + n'$ and discharge the obligation;
  - for the second obligation, any ns satisfying $\mathit{total\text{-}value\text{-}alg}\ \mathit{ns} \equiv \mathit{value}\ d + n'$ must be of the form $(\text{`}\mathsf{cons}\ , e\ , e{\leqslant}c\ , m'\ , \blacksquare)$ for some $e : \mathsf{Coin}$, $e{\leqslant}c : e \leqslant_{\mathsf{C}} c$, and $m' : \mathsf{Nat}$ since the right-hand side $\mathit{value}\ d + n'$ of the equality is non-zero, and $Q$ maps $\mathit{ns}$ to $(\text{`}\mathsf{cons}\ , d\ , d{\leqslant}c\ , n'\ , \blacksquare)$ if $e \leqslant_{\mathsf{C}} d$, so d should be the largest "usable" coin if this obligation is to be discharged. We say that $d : \mathsf{Coin}$ is **usable** with respect to some $c : \mathsf{Coin}$ and $n : \mathsf{Nat}$ if d is bounded above by c and can be part of a solution to the problem for n pence:

    $$\begin{aligned}
    &\mathit{UsableCoin} : \mathsf{Nat} \to \mathsf{Coin} \to \mathsf{Coin} \to \mathsf{Set} \\
    &\mathit{UsableCoin}\ n\ c\ d = (d \leqslant_{\mathsf{C}} c) \times (\Sigma[\, n' : \mathsf{Nat}\,]\ \mathit{value}\ d + n' \equiv n)
    \end{aligned}$$

    The obligation can then be rewritten as

    $$(e : \mathsf{Coin}) \to \mathit{UsableCoin}\ (\mathit{value}\ d + n')\ c\ e \to e \leqslant_{\mathsf{C}} d$$

which requires that d is the largest usable coin with respect to c and *value* $d + n'$. This obligation is the only one that cannot be trivially discharged, since it requires computation of the largest usable coin.

We thus specialise gcons to

```
gcons′ : {c : Coin} (d : Coin) → d ≤C c →
         {n′ : Nat} →
         ((e : Coin) → UsableCoin (value d + n′) c e → e ≤C d) →
         GreedyBag d n′ → GreedyBag c (value d + n′)
```

Because of gcons′, we are directed to implement a function *maximum-coin* that computes the largest usable coin with respect to any c : Coin and non-zero n : Nat:

```
maximum-coin :
  (c : Coin) (n : Nat) → n > 0 →
  Σ[ d : Coin ] UsableCoin n c d × ((e : Coin) → UsableCoin n c e → e ≤C d)
```

This takes some theorem proving but is overall a typical Agda exercise in dealing with natural numbers and ordering. Finally, the greedy algorithm is implemented as the following internalist program, which repeatedly uses *maximum-coin* to find the largest usable coin and unfolds a GreedyBag:

```
greedy : (c : Coin) (n : Nat) → GreedyBag c n
greedy c n = <-rec P f n c
  where
    P : Nat → Set
    P n = (c : Coin) → GreedyBag c n
    f : (n : Nat) → ((n′ : Nat) → n′ < n → P n′) → P n
    f n              rec c with compare-with-zero n
    f .0             rec c | is-zero = gnil′
    f n              rec c | above-zero n>z with maximum-coin c n n>z
    f .(value d + n′) rec c | above-zero n>z | d , (d≤c , n′ , refl) , guc =
                                                gcons′ d d≤c guc (rec n′ { }8 d)
```

In *greedy*, the combinator

```
<-rec : (P : Nat → Set) →
        ((n : Nat) → ((n′ : Nat) → n′ < n → P n′) → P n) →
        (n : Nat) → P n
```

is for well-founded recursion on $\_{<}\_$, and the function

```
compare-with-zero : (n : Nat) → ZeroView n
```

is a covering function for the view

```
data ZeroView : Nat → Set where
  is-zero    : ZeroView 0
  above-zero : {n : Nat} → n > 0 → ZeroView n
```

At Goal 8, Agda deduces that n is $\mathit{value}\ d + n'$ and demands that we prove $n' < \mathit{value}\ d + n'$ in order to make the recursive call, which is easily discharged since $\mathit{value}\ d > 0$.

5.4 Discussion

Section 5.1 heavily borrows techniques from the AoPA (Algebra of Programming in Agda) library [Mu et al., 2009] while making generalisations and adaptations: AoPA deals with non-dependently typed programs only, whereas to work with indexed datatypes we need to move to indexed families of relations; to work with the ornamental universe, we parametrise the relational fold with a description, making it fully datatype-generic, whereas AoPA has only specialised versions for lists and binary trees; we define $\mathit{min}\_ \bullet \Lambda\_$ as a single operator (which happens to be the "shrinking" operator proposed by Mu and Oliveira [2012]) to avoid the struggle with predicativity that AoPA had. All of the above are not fundamental differences between the work presented in this chapter and AoPA, though — the two differ essentially in methodology: in AoPA, dependent types merely provide a foundation on which relational program derivations can be expressed formally and checked by machine, but the programs remain non-dependently typed throughout the formalisation; whereas in this chapter, relational programming is a tool for obtaining nontrivial inductive families that effectively guide program development as advertised as the strength of internalist programming. In short, the focus of AoPA is on traditional relational program derivation (expressed in a dependently typed language), whereas our emphasis is on internalist programming (aided by relational programming).

Algebraic ornamentation was originally proposed by McBride [2011], which deals with functions only. Generalising algebraic ornamentation to a relational setting allows us to write more specifications (like the one given by the Streaming Theorem in Section 5.3.2, which involves converses) and employ more powerful theorems (like the Greedy Theorem in Section 5.3.3, which involves minimisation). In particular, since relational coalgebras coincide with the converses of algebras, we are no longer tied to the bottom-up view of datatype indices and can switch to the top-down view — for example, in the internalist implementation of the Streaming Theorem, we can think of the output list as being unfolded from the index of its type. We will show in **??** that the generalisation is in fact a "maximal" one: any datatype obtained via ornamentation can be obtained via relational algebraic ornamentation up to isomorphism. Atkey et al. [2012] also investigated algebraic ornamentation via a fibrational, syntax-free perspective. While their "partial refinement" (which generalises functional algebraic ornamentation) is subsumed by relational algebraic ornamentation (since relational algebras allow partiality), they were able to go beyond inductive families to indexed inductive-recursive datatypes [Dybjer and Setzer, 2006], which are out of the scope of this dissertation.

Let us contemplate the interplay between internalist programming and relational programming, especially the one in Section 5.3.3. As mentioned in **??**, internalist programs can encode more semantic information, including correctness proofs; we can thus write programs that directly explain their own meaning. The internalist program *greedy* is such an example, whose structure carries an implicit inductive proof; the program constructs not merely a list of coins, but a bag of coins chosen according to a particular local optimisation strategy (i.e., $\mathit{min}\ Q \bullet \Lambda\ (\mathit{fun}\ \mathit{total\text{-}value\text{-}alg}^{\circ})$). Internalist programming alone has limited power, however, because internalist programs should share structure with their correctness proofs, but we cannot expect to have such coincidences all the time. In particular, there is no hope of integrating a correctness proof into a program when the structure of the proof is more complicated than that of the program. For example, it is hard to imagine how to integrate a correctness proof for the full specification of the minimum coin change problem into a program for the greedy algorithm. In essence, we have two kinds of proofs to deal with: the first kind follow program structure and can be embedded in internalist programs, and the second kind are general proofs of full specifications, which do not necessarily follow program structure and thus fall outside the scope of internalism. To exploit the power of internalism as much as possible, we need ways to reduce the second kind of proof obligation to the first kind — note that such reduction involves not only constructing proof transformations but also determining what internalist proofs are sufficient for establishing proofs of full specifications. It turns out that relational program derivation is precisely a way in which to construct such proof transformations systematically from specifications. In relational program derivation, we identify important forms of relational programs (i.e., relational composition, recursion schemes, and various other combinators), and formulate algebraic laws and theorems in terms of these forms. By applying the laws and theorems, we massage a relational specification into a known form which corresponds to a proof obligation that can be expressed in an internalist type, enabling transition to internalist programming. For example, we now know that a relational fold can be turned into an inductive family for internalist programming by algebraic ornamentation. Thus, given a relational specification, we might seek to massage it into a relational fold when that possibility is pointed out by known laws and theorems (e.g., the Greedy Theorem). To sum up, we get a hybrid methodology that leads us from relational specifications towards internalist types for type-directed programming, providing hints on internalist type design in the form of relational algebraic laws and theorems.

Bibliography

Thorsten ALTENKIRCH, James CHAPMAN, and Tarmo UUSTALU [2010]. Monads need not be endofunctors. In *Foundations of Software Science and Computational Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 297–311. Springer-Verlag. doi: 10.1007/978-3-642-12032-9_21. ↰ page 3

Robert ATKEY, Patricia JOHANN, and Neil GHANI [2012]. Refining inductive types. *Logical Methods in Computer Science*, 8(2:9). doi: 10.2168/LMCS-8(2:9)2012. ↰ page 34

John BACKUS [1978]. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641. doi: 10.1145/359576.359579. ↰ page 2

Richard BIRD [1996]. Functional algorithm design. *Science of Computer Programming*, 26(1–3):15–31. doi: 10.1016/0167-6423(95)00033-X. ↰ page 2

Richard BIRD [2010]. *Pearls of Functional Algorithm Design*. Cambridge University Press. ISBN: 978-0521513388. ↰ page 2

Richard BIRD and Oege DE MOOR [1997]. *Algebra of Programming*. Prentice-Hall. ISBN: 978-0135072455. ↰ pages 1, 6, 10, 23, and 24

Richard BIRD and Jeremy GIBBONS [2003]. Arithmetic coding with folds and unfolds. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag. doi: 10.1007/978-3-540-44833-4_1. ↰ pages 10, 15, and 20

Peter Dybjer and Anton Setzer [2006]. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66(1):1–49. doi: `10.1016/j.jlap.2005.07.001`. ↰ page 34

Jeremy Gibbons [2007]. Metamorphisms: Streaming representation-changers. *Science of Computer Programming*, 65(2):108–139. doi: `10.1016/j.scico.2006.01.006`. ↰ pages 15 and 19

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. URL: `https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf`. ↰ pages 9 and 34

Conor McBride and Ross Paterson [2008]. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13. doi: `10.1017/S0956796807006326`. ↰ page 3

Erik Meijer, Maarten Fokkinga, and Ross Paterson [1991]. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 124–144. Springer-Verlag. doi: `10.1007/3540543961_7`. ↰ page 6

Eugenio Moggi [1991]. Notions of computation and monads. *Information and Computation*, 93(1):55–92. doi: `10.1016/0890-5401(91)90052-4`. ↰ page 3

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson [2009]. Algebra of Programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19(5):545–579. doi: `10.1017/S0956796809007345`. ↰ pages 14 and 33

Shin-Cheng Mu and José Nuno Oliveira [2012]. Programming from Galois connections. *Journal of Logic and Algebraic Programming*, 81(6):680–704. doi: `10.1016/j.jlap.2012.05.003`. ↰ page 33

Keisuke Nakano [2013]. Metamorphism in jigsaw. *Journal of Functional Programming*, 23(2):161–173. doi: `10.1017/S0956796812000391`. ↰ page 15

Bengt Nordström [1988]. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619. doi: `10.1007/BF01941137`. ↰ page 20

Philip Wadler [1992]. The essence of functional programming. In *Principles of Programming Languages*, POPL'92, pages 1–14. ACM. doi: `10.1145/143165.143169`. ↰ page 3

Todo list