Chapter 5

Relational algebraic ornaments

5.1 Relational program derivation in Agda and relational algebraic ornamentation

In this section, we first introduce and formalise some basic notions in relational program derivation Bird and de Moor [1997] by importing and generalising a small part of the AoPA library Mu et al. [2009]. We then introduce *relational algebraic ornamentation*, which acts as a bridge between the two worlds of internalist programming and relational program derivation. At the end of this section is an example about the *Fold Fusion Theorem* [Bird and de Moor, 1997, Section 6.2] and how the theorem translates to conversion functions between algebraically ornamented datatypes.

Basic definitions for relational program derivation. One common approach to program derivation is by algebraic transformations of functional programs: one begins with a specification in the form of a functional program that expresses 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 would enforce a particular way of choosing the optimal solution, but such enforcement should not be part of the specification. To gain more flexibility, the specification language was later generalised to *relational programs*. 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 element of A to a *subset* of B. We define subsets by

$$\wp : \mathsf{Set} \to \mathsf{Set}_1$$
$$(\mathit{Power}\ A) = A \to \mathsf{Set}$$

That is, a subset $s : (\mathit{Power}\ A)$ is a characteristic function that assigns a type to each element 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 $(\mathit{Power}\ A)$ as the type of computations that nondeterministically produce an element of A. A simple example is

$$\mathit{any} : \{A : \mathsf{Set}\} \to (\mathit{Power}\ A)$$
$$\mathit{any} = \mathit{const}\ \top$$

The subset $\mathit{any} : (\mathit{Power}\ A)$ associates the unit type $\top$ with every element of A. Since $\top$ is inhabited, *any* can produce any element of A. $\wp$ cannot be made into a conventional monad because it is not an endofunctor, but it still has a monadic structure Altenkirch et al. [2010]: *return* and $\_ >>= \_$ are defined as

$$\mathit{return} : \{A : \mathsf{Set}\} \to A \to (\mathit{Power}\ A)$$
$$\mathit{return} = \_{\equiv}\_$$

$$\_ >>= \_ : \{A\ B : \mathsf{Set}\} \to (\mathit{Power}\ A) \to (A \to (\mathit{Power}\ B)) \to (\mathit{Power}\ B)$$

$$\_ >>= \_ \; \{A\} \; s \, f \; = \; \lambda \; b \rightarrow \Sigma \langle \, a : A \, \rangle \;\; s \; a \times f \; a \; b$$

The subset *return* $a \; : \; (Power \; A)$ for some $a \; : \; A$ simplifies to $\lambda \; a' \rightarrow a \; \equiv \; a'$ (where $\_{\equiv}\_$ is propositional equality), so a is the only member of the subset; if $s \; : \; (Power \; A)$ and $f \; : \; A \rightarrow (Power \; B)$, then the subset $s \; \gg\!= \; f \; : \; (Power \; B)$ is the union of all the subsets $f \; a \; : \; (Power \; B)$ where a ranges over the elements of A that belong to s, i.e., an element $b \; : \; B$ 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 will mainly use relations between families of sets in this paper: if $X, Y \; : \; I \rightarrow \mathsf{Set}$ for some $I \; : \; \mathsf{Set}$, a relation from X to Y is defined as a family of relations from $X \; i$ to $Y \; i$ for every $i \; : \; I$.

$$\begin{aligned}
&\_ \leadsto \_ \; : \; \{I \; : \; \mathsf{Set}\} \rightarrow (I \rightarrow \mathsf{Set}) \rightarrow (I \rightarrow \mathsf{Set}) \rightarrow \mathsf{Set}_1 \\
&X \leadsto Y \; = \; \forall \; \{i\} \rightarrow X \; i \rightarrow (Power \; (Y \; i))
\end{aligned}$$

We can use the subset combinators to define relations. For example, the following combinator *fun* lifts a family of functions into a family of relations.

$$\begin{aligned}
&fun \; : \; \{I \; : \; \mathsf{Set}\} \; \{X \; Y \; : \; I \rightarrow \mathsf{Set}\} \rightarrow (X \rightrightarrows Y) \rightarrow (X \leadsto Y) \\
&fun \, f \; x \; = \; return \; (f \; x)
\end{aligned}$$

The identity relation is just the identity functions lifted to relations.

$$\begin{aligned}
&idR \; : \; \{I \; : \; \mathsf{Set}\} \; \{X \; : \; I \rightarrow \mathsf{Set}\} \rightarrow (X \leadsto X) \\
&idR \; = \; fun \; id
\end{aligned}$$

Composition of relations is easily defined with $\_ >>= \_$: computing $R \; \cdot \; S$ on input x is first computing $S \; x$ and then feeding the result to R.

$$\begin{aligned}
&\_\mathrm{ffl}\_ \; : \; \{I \; : \; \mathsf{Set}\} \; \{X \; Y \; Z \; : \; I \rightarrow \mathsf{Set}\} \rightarrow \\
&\qquad\quad (Y \leadsto Z) \rightarrow (X \leadsto Y) \rightarrow (X \leadsto Z) \\
&(R \; \cdot \; S) \; x \; = \; S \; x \; \gg\!= \; R
\end{aligned}$$

Or we may choose to define a relation pointwise, like

$$\begin{aligned}
&\_ \cap \_ \; : \; \{I \; : \; \mathsf{Set}\} \; \{X \; Y \; : \; I \rightarrow \mathsf{Set}\} \rightarrow \\
&\qquad\quad (X \leadsto Y) \rightarrow (X \leadsto Y) \rightarrow (X \leadsto Y) \\
&(R \cap S) \; x \; y \; = \; R \; x \; y \times S \; x \; y
\end{aligned}$$

This defines the meet of two relations. 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:

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

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 Section 5.2.

Laws and theorems in relational program derivation are formulated with *relational inclusion*

$$
\begin{aligned}
&\_\subseteq\_ \;:\; \{I \;:\; \mathsf{Set}\}\; \{X\, Y \;:\; I \to \mathsf{Set}\}\; (R\, S \;:\; X \leadsto Y) \to \mathsf{Set}\\
&R \subseteq S \;=\; \forall\, \{i\} \to (x \;:\; X\, i)\; (y \;:\; Y\, i) \to R\, x\, y \to S\, x\, y
\end{aligned}
$$

or equivalence of relations, which is defined as two-way inclusion:

$$
\begin{aligned}
&\_\simeq\_ \;:\; \{I \;:\; \mathsf{Set}\}\; \{X\, Y \;:\; I \to \mathsf{Set}\}\; (R\, S \;:\; X \leadsto Y) \to \mathsf{Set}\\
&R \simeq S \;=\; (R \subseteq S) \times (S \subseteq R)
\end{aligned}
$$

We will also need *relators*, i.e., monotonic functors on relations with respect to relational inclusion.

$$
\begin{aligned}
&\mathbb{R} \;:\; \{I \;:\; \mathsf{Set}\}\; (D \;:\; \mathsf{Desc}\; I)\; \{X\, Y \;:\; I \to \mathsf{Set}\} \to\\
&\qquad (X \leadsto Y) \to (\mathbb{F}\; D\; X \leadsto \mathbb{F}\; D\; Y)
\end{aligned}
$$

If $R \;:\; X \leadsto Y$, the relation $\mathbb{R}\; D\; R \;:\; \mathbb{F}\; D\; X \leadsto \mathbb{F}\; D\; Y$ applies R to the recursive positions of its input, leaving everything else intact. For example, if $D \;=\; \mathit{ListD}\; A$ (for some $A \;:\; \mathsf{Set}$), then $\mathbb{R}\; (\mathit{ListD}\; A)$ essentially specialises to

$$
\begin{aligned}
&\mathbb{R}\; (\mathit{ListD}\; A) \;:\; \{X\, Y \;:\; I \to \mathsf{Set}\} \to\\
&\qquad\qquad (X \leadsto Y) \to (\mathbb{F}\; (\mathit{ListD}\; A)\; X \leadsto \mathbb{F}\; (\mathit{ListD}\; A)\; Y)\\
&\mathbb{R}\; (\mathit{ListD}\; A)\; R\; (\mathit{nil}-\mathit{tag}\quad , \blacksquare) \qquad = \mathit{return}\; (\mathit{nil}-\mathit{tag}\; , \blacksquare)\\
&\mathbb{R}\; (\mathit{ListD}\; A)\; R\; (\mathit{cons}-\mathit{tag}\; , a\; , x) \;=\; R\; x \ggg\!= \lambda\; y \to \mathit{return}\; (\mathit{cons}-\mathit{tag}\; , a\; , y)
\end{aligned}
$$

Among other properties, we can prove that $\mathbb{R}\; D$ preserves identity ($\mathbb{R}\; D\; \mathit{idR} \simeq \mathit{idR}$), composition ($\mathbb{R}\; D\; (R \cdot S) \simeq \mathbb{R}\; D\; R \cdot \mathbb{R}\; D\; S$), converse ($\mathbb{R}\; D\; (R^{\circ}) \simeq (\mathbb{R}\; D\; R)^{\circ}$), and is monotonic ($R \subseteq S$ implies $\mathbb{R}\; D\; R \subseteq \mathbb{R}\; D\; S$).

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

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

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. In the context of optimisation problems, monotonicity can be used to capture the *principle of optimality*, as will be shown in Section 5.2.

Having defined relations as nondeterministic mappings, it is straightforward to port the datatype-generic *fold* to relations:

$$\begin{aligned}&\mathit{foldR} : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\}\ \{X : I \rightarrow \mathsf{Set}\} \rightarrow\\ &\qquad (\mathbb{F}\ D\ X \rightsquigarrow X) \rightarrow (\mu\ D \rightsquigarrow X)\end{aligned}$$

The definition of *foldR* is obtained by rewriting the definition of *fold* with the subset combinators. For example, the relational fold on lists would essentially be

$$\begin{aligned}&\mathit{foldR}\ \{\top\}\ \{\mathit{ListD}\ A\} &&: \{X : \top \rightarrow \mathsf{Set}\} \rightarrow\\ & && \quad (\mathbb{F}\ (\mathit{ListD}\ A)\ X \rightsquigarrow X) \rightarrow\\ & && \quad (\mu\ (\mathit{ListD}\ A) \quad \rightsquigarrow X)\\ &(\mathit{cata}\ (R))\ [\,] &&= R\ (\mathit{nil} - \mathit{tag}\ ,\ \blacksquare)\\ &(\mathit{cata}\ (R))\ (a :: \mathit{as}) &&= (\mathit{cata}\ (R))\ \mathit{as} \ggg\!= \lambda\ x \rightarrow R\ (\mathit{cons} - \mathit{tag}\ ,\ a\ ,\ x)\end{aligned}$$

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

$$\begin{aligned}&\mathit{fun} - \mathit{preserves} - \mathit{fold}\ :\\ &\quad \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ \{X : I \rightarrow \mathsf{Set}\}\\ &\quad (f : \mathbb{F}\ D\ X \rightrightarrows X) \rightarrow \mathit{fun}\ (\mathit{fold}\ f) \simeq (\mathit{cata}\ (\mathit{fun}\ f))\end{aligned}$$

Relational algebraic ornamentation. We now turn to relational algebraic ornamentation, the key construct that bridges internalist programming and relational program derivation. Let $R : \mathbb{F}\ (ListD\ A)\ X \rightsquigarrow X$ (where $X : \top \to \mathsf{Set}$) be a relational algebra for lists. We can define a datatype of "algebraic lists" as

```
indexfirst data AlgList A R : X ▪ → Set where
  AlgList A R x accepts nil (rnil : R (nil − tag , ▪) x)
                     |      cons (a : A) (x′ : X ▪) (as : AlgList A R x′)
                                 (rcons : R (cons − tag , a , x′) x)
```

There is an ornament from lists to algebraic lists which marks the fields *rnil*, x', and *rcons* in *AlgList* as additional and refines the index of the recursive position to x'. The promotion predicate for this ornament is

```
indexfirst data AlgListP A R : X ▪ → List A → Set where
  AlgListP A R x []       accepts nil (rnil : R (nil − tag , ▪) x)
  AlgListP A R x (a :: as) accepts cons (x′ : X ▪)
                                        (p : AlgListP A R x′ as)
                                        (rcons : R (cons − tag , a , x′) x)
```

A simple argument by induction shows that $AlgListP\ A\ R\ x\ as$ is in fact isomorphic to $(cata\ (R))\ as\ x$ for any $as : \mathsf{List}\ A$ and $x : X\ ▪$. As a corollary, we have

$$AlgList\ A\ R\ x \;\cong\; \Sigma\langle\, as : \mathsf{List}\ A\,\rangle\;\; (cata\ (R))\ as\ x \tag{5.1}$$

for any $x : X\ ▪$ by (**??**). That is, an algebraic list is exactly a plain list and a proof that the list folds to x using the algebra R. The vector datatype is a special case of *AlgList* — to see that, define

```
length − alg : 𝔽 (ListD A) (const Nat) ⇉ const Nat
length − alg (nil − tag   , ▪)     = zero
length − alg (cons − tag , a , n)  = suc n
```

and take $R = fun\ length - alg$. From (5.1) we have the isomorphisms

$$\mathsf{Vec}\ A\ n \;\cong\; \Sigma\langle\, as : \mathsf{List}\ A\,\rangle\;\; (cata\ (fun\ length - alg))\ as\ n$$

for all $n : \mathsf{Nat}$, from which we can derive

$$\mathsf{Vec}\ A\ n \;\cong\; \Sigma\langle\, as : \mathsf{List}\ A\,\rangle\;\; length\ as \;\equiv\; n$$

by $\mathit{fun}-\mathit{preserves}-\mathit{fold}$, after defining $\mathit{length} = \mathit{fold}\ \mathit{length}-\mathit{alg}$.

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

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

(where $\mathsf{outl} : \Sigma\ I\ X \to I$). Its definition is a slight generalisation of the one given by Dagand and McBride [Dagand and McBride, 2012, supplementary code]. The promotion predicate for the ornament $\lceil \mathit{algOrn}\ D\ R \rceil$ is pointwise isomorphic to $(\mathit{cata}\ (R))$, i.e.,

$$\mathit{PromP}\ \lceil \mathit{algOrn}\ D\ R \rceil\ (\mathsf{ok}\ (i\ ,x))\ d\ \cong\ (\mathit{cata}\ (R))\ d\ x \tag{5.2}$$

for all $i : I$, $x : X\ i$, and $d : \mu\ D\ i$. As a corollary, we have the following isomorphisms

$$\mu\ \lfloor \mathit{algOrn}\ D\ R \rfloor\ (i\ ,x)\ \cong\ \Sigma \langle\, d : \mu\ D\ i \,\rangle\ (\mathit{cata}\ (R))\ d\ x \tag{5.3}$$

for all $i : I$ and $x : X\ i$ by (**??**). For example, taking $D = \mathit{ListD}\ A$, the type $\mathit{AlgList}\ A\ R\ x$ can be thought of as the high-level presentation of $\mu\ \lfloor \mathit{algOrn}\ (\mathit{ListD}\ A)\ R \rfloor\ (\blacksquare\ ,x)$. Algebraic ornamentation is a very convenient method for adding new indices to inductive families, and most importantly, it says precisely what the new indices mean. The method was demonstrated by McBride [2011] with a correct-by-construction compiler for a small language, and will be demonstrated again in Section 5.2.

Example: the Fold Fusion Theorem. As a first example of bridging internalist programming with relational program derivation through algebraic ornamentation, let us consider the *Fold Fusion Theorem* [Bird and de Moor, 1997, Section 6.2]: Let $D : \mathsf{Desc}\ I$ be a description, $R : X \rightsquigarrow Y$ a relation, and $S : \mathbb{F}\ D\ X \rightsquigarrow X$ and $T : \mathbb{F}\ D\ Y \rightsquigarrow Y$ be algebras. If R is a homomorphism from S to T, i.e.,

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

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

$$R \cdot (\mathit{cata}\ (S)) \simeq (\mathit{cata}\ (T))$$

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 of the variations is

$$R \cdot S \subseteq T \cdot \mathbb{R}\, D\, R \quad \text{implies} \quad R \cdot (\mathit{cata}\ (S)) \subseteq (\mathit{cata}\ (T))$$

This can be used with (5.3) to derive a conversion between algebraically ornamented datatypes:

$$\begin{array}{l}
\mathit{algOrn} - \mathit{fusion} - \subseteq D\; R\; S\; T \;: \\
\quad R \cdot S \subseteq T \cdot \mathbb{R}\, D\, R \rightarrow \\
\quad \{i : I\}\ (x : X\, i) \rightarrow \mu\ \lfloor \mathit{algOrn}\ D\ S \rfloor\ (i\,, x) \rightarrow \\
\quad (y : Y\, i) \rightarrow R\; x\; y \rightarrow \mu\ \lfloor \mathit{algOrn}\ D\ T \rfloor\ (i\,, y)
\end{array}$$

The other variation of Fold Fusion simply reverses the direction of inclusion:

$$R \cdot S \supseteq T \cdot \mathbb{R}\, D\, R \quad \text{implies} \quad R \cdot (\mathit{cata}\ (S)) \supseteq (\mathit{cata}\ (T))$$

which translates to the conversion

$$\begin{array}{l}
\mathit{algOrn} - \mathit{fusion} - \supseteq D\; R\; S\; T \;: \\
\quad R \cdot S \supseteq T \cdot \mathbb{R}\, D\, R \rightarrow \\
\quad \{i : I\}\ (y : Y\, i) \rightarrow \mu\ \lfloor \mathit{algOrn}\ D\ T \rfloor\ (i\,, y) \rightarrow \\
\quad \Sigma\langle\, x : X\, i \,\rangle\ \mu\ \lfloor \mathit{algOrn}\ D\ S \rfloor\ (i\,, x) \times R\; x\; y
\end{array}$$

For a simple example, 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

$$\begin{array}{l}
\mathit{BVec} : \mathsf{Set} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Set} \\
\mathit{BVec}\ A\ m = \\
\quad \mu\ \lfloor \mathit{algOrn}\ (\mathit{ListD}\ A)\ (\mathit{geq} \cdot \mathit{fun}\ \mathit{length} - \mathit{alg}) \rfloor\ (\blacksquare\,, m)
\end{array}$$

where $\mathit{geq} = \lambda\, x\, y \rightarrow x \leqslant y : \mathit{const}\ \mathsf{Nat} \rightsquigarrow \mathit{const}\ \mathsf{Nat}$ maps a natural number x to any natural number that is at least x. The isomorphisms (5.3) specialise for *BVec* to

$$\mathit{BVec}\ A\ m \cong \Sigma\langle\, \mathit{as} : \mathsf{List}\ A \,\rangle\ (\mathit{cata}\ (\mathit{geq} \cdot \mathit{fun}\ \mathit{length} - \mathit{alg}))\ \mathit{as}\ m$$

But is *BVec* really the bounded vectors? Indeed it is, because we can deduce

$$geq \cdot (cata\ (fun\ length - alg)) \simeq (cata\ (geq \cdot fun\ length - alg))$$

by Fold Fusion (where $(cata\ (fun\ length - alg))$ is equivalent to $fun\ length$ by $fun - preserves - fold$). The fusion condition is

$$geq \cdot fun\ length - alg \simeq geq \cdot fun\ length - alg \cdot \mathbb{R}\ (ListD\ A)\ geq$$

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

$$\begin{aligned}
& geq \cdot fun\ length - alg \\
\subseteq\ & \{\ idR \text{ identity}\ \} \\
& geq \cdot fun\ length - alg \cdot idR \\
\subseteq\ & \{\ \text{relator preserves identity}\ \} \\
& geq \cdot fun\ length - alg \cdot \mathbb{R}\ (ListD\ A)\ idR \\
\subseteq\ & \{\ geq \text{ reflexive}\ \} \\
& geq \cdot fun\ length - alg \cdot \mathbb{R}\ (ListD\ A)\ geq
\end{aligned}$$

And from right to left:

$$\begin{aligned}
& geq \cdot fun\ length - alg \cdot \mathbb{R}\ (ListD\ A)\ geq \\
\subseteq\ & \{\ fun\ length - alg \text{ monotonic on } geq\ \} \\
& geq \cdot geq \cdot fun\ length - alg \\
\subseteq\ & \{\ geq \text{ transitive}\ \} \\
& geq \cdot fun\ length - alg
\end{aligned}$$

Note that these calculations are good illustrations of the power of relational calculation despite their simplicity — they are straightforward symbolic 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 $algOrn - fusion - \subseteq$ and $algOrn - fusion - \supseteq$. Hence we get two conversions, one of type

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

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

$BVec\ A\ m \rightarrow \Sigma \langle n : \mathsf{Nat} \rangle\ \mathsf{Vec}\ A\ n \times (n \leqslant m)$

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.

Theoretically, the conversions derived from Fold Fusion are actually more generally applicable than they seem, because *every ornament is an algebraic ornament up to isomorphism*. This we show next.

5.2 Example: 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:

$$\begin{array}{l} \textbf{data}\ Coin\ :\ \mathsf{Set}\ \textbf{where} \\ \quad onep\ twop\ fivep\ :\ Coin \end{array}$$

Given $n\ :\ \mathsf{Nat}$, the *minimum coin change problem* asks for the least number of coins that make up n pence. We can give a relational specification of the problem with the following operator:

$$\begin{array}{l} min\_ \cdot \Lambda\_\ :\ \{I\ :\ \mathsf{Set}\}\ \{X\ Y\ :\ I \rightarrow \mathsf{Set}\} \\ \qquad\qquad (R\ :\ Y \rightsquigarrow Y)\ (S\ :\ X \rightsquigarrow Y) \rightarrow (X \rightsquigarrow Y) \\ (min\ R\ \cdot \Lambda\ S)\ x\ y\ =\ S\ x\ y \times (\forall\ y' \rightarrow S\ x\ y' \rightarrow R\ y'\ y) \end{array}$$

An input $x\ :\ X\ i$ for some $i\ :\ I$ is mapped by $min\ R\ \cdot \Lambda\ S$ to $y\ :\ Y\ i$ if y is a possible result in $S\ x\ :\ (Power\ (Y\ i))$ and is the smallest such result under R, in the sense that any y' in $S\ x\ :\ (Power\ (Y\ i))$ must satisfy $R\ y'\ y$ (i.e., R maps larger inputs to smaller outputs). Intuitively, we can think of $min\ R\ \cdot \Lambda\ S$ as consisting of two steps: the first step $\Lambda\ S$ computes the set of all possible results yielded by S, and the second step $min\ R$ chooses a minimum result from that set (nondeterministically). 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 turn to this representation having attempted to apply the

Greedy Theorem, which will be introduced shortly.) If we define the ordering on coins as

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

where the values of the coins are defined by

$$\begin{aligned}
&\mathit{value} \;:\; \mathit{Coin} \to \mathsf{Nat}\\
&\mathit{value}\;\mathit{onep} \;=\; 1\\
&\mathit{value}\;\mathit{twop} \;=\; 2\\
&\mathit{value}\;\mathit{fivep} \;=\; 5
\end{aligned}$$

then the datatype of coin bags we use is

$$\begin{aligned}
&\textbf{indexfirst data}\; \mathit{CoinBag} \;:\; \mathit{Coin} \to \mathsf{Set}\;\textbf{where}\\
&\quad \mathit{CoinBag}\; c\; \mathit{accepts}\; \mathsf{nil}\\
&\qquad\qquad\qquad\quad \mid \quad \mathsf{cons}\; (d \;:\; \mathit{Coin})\; (\mathit{leq} \;:\; d \leqslant C\; c)\; (b \;:\; \mathit{CoinBag}\; d)
\end{aligned}$$

Down at the universe level, the (ornamental) description of *CoinBag* (relative to List *Coin*) is simply that of OrdList *Coin* (*flip* _ ⩽ *C*_).

$$\begin{aligned}
&\mathit{CoinBagOD} \;:\; \mathsf{OrnDesc}\; \mathit{Coin}\; !\; (\mathit{ListD}\; \mathit{Coin})\\
&\mathit{CoinBagOD} \;=\; \mathit{OrdListOD}\; \mathit{Coin}\; (\mathit{flip}\; \_\leqslant C\_)\\
&\mathit{CoinBagD} \;:\; \mathsf{Desc}\; \mathit{Coin}\\
&\mathit{CoinBagD} \;=\; \lfloor \mathit{CoinBagOD} \rfloor\\
&\mathit{CoinBag} \;:\; \mathit{Coin} \to \mathsf{Set}\\
&\mathit{CoinBag} \;=\; \mu\; \mathit{CoinBagD}
\end{aligned}$$

The base functor for *CoinBag* is

$$\begin{aligned}
&\mathbb{F}\; \mathit{CoinBagD} \;:\; (\mathit{Coin} \to \mathsf{Set}) \to (\mathit{Coin} \to \mathsf{Set})\\
&\mathbb{F}\; \mathit{CoinBagD}\; X\; c \;=\\
&\quad \Sigma\; \mathit{LTag}\; (\lambda\; \{\mathit{nil}-\mathit{tag} \to \top; \mathit{cons}-\mathit{tag} \to \Sigma\langle\, d : \mathit{Coin}\,\rangle\;\; (d \leqslant C\; c) \times X\; d\})
\end{aligned}$$

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:

$$\begin{aligned}
&\mathit{total}-\mathit{value}-\mathit{alg} \;:\; \mathbb{F}\; \mathit{CoinBagD}\; (\mathit{const}\; \mathsf{Nat}) \rightrightarrows \mathit{const}\; \mathsf{Nat}\\
&\mathit{total}-\mathit{value}-\mathit{alg}\; (\mathit{nil}-\mathit{tag} \quad , \_ \qquad\quad) \;=\; 0
\end{aligned}$$

$$
\begin{aligned}
&\mathit{total-value-alg}\ (\mathit{cons-tag}\ ,\ d\ ,\ \_\ ,\ n) \ = \ \mathit{value}\ d + n\\
&\mathit{total-value}\ :\ \mathit{CoinBag} \rightrightarrows \mathit{const}\ \mathsf{Nat}\\
&\mathit{total-value}\ =\ \mathit{fold}\ \mathit{total-value-alg}
\end{aligned}
$$

and the number of coins in a coin bag is also computed by a fold:

$$
\begin{aligned}
&\mathit{size-alg}\ :\ \mathbb{F}\ \mathit{CoinBagD}\ (\mathit{const}\ \mathsf{Nat}) \rightrightarrows \mathit{const}\ \mathsf{Nat}\\
&\mathit{size-alg}\ (\mathit{nil-tag}\quad ,\ \_\qquad\quad)\ =\ 0\\
&\mathit{size-alg}\ (\mathit{cons-tag}\ ,\ \_\ ,\ \_\ ,\ n)\ =\ 1 + n\\
&\mathit{size}\ :\ \mathit{CoinBag} \rightrightarrows \mathit{const}\ \mathsf{Nat}\\
&\mathit{size}\ =\ \mathit{fold}\ \mathit{size-alg}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{min-coin-change}\ :\ \mathit{const}\ \mathsf{Nat} \rightsquigarrow \mathit{CoinBag}\\
&\mathit{min-coin-change}\ =\\
&\quad \mathit{min}\ (\mathit{fun}\ \mathit{size}^{\,\mathrm{o}}\ \cdot\ \mathit{leq}\ \cdot\ \mathit{fun}\ \mathit{size})\ \cdot \Lambda\ (\mathit{fun}\ \mathit{total-value}^{\,\mathrm{o}})
\end{aligned}
$$

where $\mathit{leq} = \mathit{geq}^{\,\mathrm{o}} : \mathit{const}\ \mathsf{Nat} \rightsquigarrow \mathit{const}\ \mathsf{Nat}$ maps a natural number n to any natural number that is at most n. Intuitively, given an input $n : \mathsf{Nat}$, the relation $\mathit{fun}\ \mathit{total-value}^{\,\mathrm{o}}$ computes an arbitrary coin bag whose total value is n, so $\mathit{min-coin-change}$ 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 $f : \mathit{const}\ \mathsf{Nat} \rightrightarrows \mathit{CoinBag}$ such that $\mathit{fun}\ f \subseteq \mathit{min-coin-change}$, and then $f\ \mathit{fivep} : \mathsf{Nat} \to \mathit{CoinBag}\ \mathit{fivep}$ would be a solution — note that the type $\mathit{CoinBag}\ \mathit{fivep}$ contains all coin bags, since fivep 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, we would attempt to refine $\mathit{min-coin-change}$ 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 we can derive that, for some algebra $R : \mathbb{F}\ \mathit{CoinBagD}\ (\mathit{const}\ \mathsf{Nat}) \rightsquigarrow \mathit{const}\ \mathsf{Nat}$,

$$(\mathit{cata}\ (R))\ ^{\mathrm{o}} \subseteq \mathit{min} - \mathit{coin} - \mathit{change} \tag{5.4}$$

then we can manufacture a new datatype

```
GreedySolutionOD : OrnDesc (Coin × Nat) outl CoinBagD
GreedySolutionOD = algOrn CoinBagD R

GreedySolution : Coin → Nat → Set
GreedySolution c n = μ ⌊GreedySolutionOD⌋ (c , n)
```

and construct a function of type

```
greedy : (c : Coin) (n : Nat) → GreedySolution c n
```

from which we can assemble a solution

```
sol : Nat → CoinBag fivep
sol = forget ⌈GreedySolutionOD⌉ ∘ greedy fivep
```

The program *sol* satisfies the specification because of the following argument: For any $c : \mathit{Coin}$ and $n : \mathsf{Nat}$, by (5.3) we have

$$\mathit{GreedySolution}\ c\ n \ \cong\ \Sigma\langle\, b : \mathit{CoinBag}\ c\,\rangle\ \ (\mathit{cata}\ (R))\ b\ n$$

In particular, since the first half of the left-to-right direction of the isomorphism is $\mathit{forget}\ \lceil \mathit{GreedySolutionOD} \rceil$, we have

$$(\mathit{cata}\ (R))\ (\mathit{forget}\ \lceil \mathit{GreedySolutionOD} \rceil\ g)\ n$$

for any $g : \mathit{GreedySolution}\ c\ n$. Substituting g by $\mathit{greedy}\ \mathit{fivep}\ n$, we get

$$(\mathit{cata}\ (R))\ (\mathit{sol}\ n)\ n$$

which implies, by (5.4),

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

i.e., *sol* satisfies the specification. Thus all we need to do to solve the minimum coin change problem is (i) refine the specification $\mathit{min} - \mathit{coin} - \mathit{change}$ to the converse of a fold, i.e., find the algebra R in (5.4), and (ii) 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 essentially a specialisation of Bird and de Moor's Theorem 10.1 Bird and de Moor [1997]: 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

$$min\ R\ \cdot \Lambda\ ((cata\ (S))\ ^{\mathrm{o}})$$

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 $(cata\ (S))$. The Greedy Theorem states that, if the initial algebra $\alpha\ =\ fun\ \mathsf{con}\ :\ \mathbb{F}\ D\ (\mu\ D) \rightsquigarrow \mu\ D$ is monotonic on R (where $\mathsf{con}\ :\ \mathbb{F}\ D\ (\mu\ D) \rightrightarrows \mu\ D$ is the datatype-generic constructor), i.e.,

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

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

$$\alpha\ \cdot\ \mathbb{R}\ D\ ((cata\ (S))\ ^{\mathrm{o}})\ \cdot\ (Q \cap (S\ ^{\mathrm{o}}\ \cdot\ S))\ ^{\mathrm{o}} \subseteq R\ ^{\mathrm{o}}\ \cdot\ \alpha\ \cdot\ \mathbb{R}\ D\ ((cata\ (S))\ ^{\mathrm{o}})$$

is satisfied, then we have

$$(cata\ ((min\ Q\ \cdot \Lambda\ (S\ ^{\mathrm{o}}))\ ^{\mathrm{o}}))\ ^{\mathrm{o}} \subseteq min\ R\ \cdot \Lambda\ ((cata\ (S))\ ^{\mathrm{o}})$$

Here we offer an intuitive explanation of the Greedy Theorem, but the theorem admits an elegant calculational proof, which can be faithfully reprised in Agda. 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 subproblems would lead to a better solution to the original problem, so the *principle of optimality* applies, i.e., to reach an optimal solution it suffices to find optimal solutions to subproblems, 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 which has significant influence on the quality of solutions: Suppose

```
data CoinBag′View : {c : Coin} {n : Nat} {l : Nat} → CoinBag′ c n l → Set where
  empty      : {c : Coin} → CoinBag′View {c} {0} {0} bnil′
  oneponep   : {m l : Nat} {lep : onep ⩽ C onep} (b : CoinBag′ onep m l) → CoinBag′View {one
  oneptwop   : {m l : Nat} {lep : onep ⩽ C twop} (b : CoinBag′ onep m l) → CoinBag′View {two
  twoptwop   : {m l : Nat} {lep : twop ⩽ C twop} (b : CoinBag′ twop m l) → CoinBag′View {tw
  onepfivep  : {m l : Nat} {lep : onep ⩽ C fivep} (b : CoinBag′ onep m l) → CoinBag′View {five
  twopfivep  : {m l : Nat} {lep : twop ⩽ C fivep} (b : CoinBag′ twop m l) → CoinBag′View {five
  fivepfivep : {m l : Nat} {lep : fivep ⩽ C fivep} (b : CoinBag′ fivep m l) → CoinBag′View {five
```

Figure 5.1 The view datatype on *CoinBag′*.

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 $S\ ^{o}\ x : (Power\ (\mathbb{F}\ D\ X\ i))$, and assume that xs is better than xs' under Q. Then for any solution resulting from xs' (computed by $\alpha\ \cdot\ \mathbb{R}\ D\ ((cata\ (S))\ ^{o}))$ 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 $min\ Q\ \cdot \Lambda\ (S\ ^{o}) : X \rightsquigarrow \mathbb{F}\ D\ X$.

Back to the coin changing problem: By $fun - preserves - fold$, the specification $min - coin - change$ is equivalent to

$$min\ (fun\ size\ ^{o}\ \cdot\ leq\ \cdot\ fun\ size)\ \cdot \Lambda\ ((cata\ (fun\ total - value - alg))\ ^{o})$$

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 possible denomination, so we go for

```
Q : 𝔽 CoinBagD (const Nat) ⇝ 𝔽 CoinBagD (const Nat)
Q (nil − tag    , _    ) = return (nil − tag , ▪)
Q (cons − tag , d , _) =
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
greedy − lemma .onep .onep _     .n      n       refl l          b (vartype (CoinBag′ onep n l)) |
greedy − lemma .onep .twop _     .(1 + n) n      refl l          b | oneptwop with view − CoinB
greedy − lemma .onep .twop _     .(1 + n) n      refl .(1 + l″) ._ | oneptwop | oneponep {.n} {l

greedy − lemma .onep .fivep _    .(4 + n) n      refl l          b | onepfivep with view − CoinB
greedy − lemma .onep .fivep _    .(4 + n) n      refl ._         ._ | onepfivep | oneponep b wit
greedy − lemma .onep .fivep _    .(4 + n) n      refl ._         ._ | onepfivep | oneponep . _ | o
greedy − lemma .onep .fivep _    .(4 + n) n      refl ._         ._ | onepfivep | oneponep . _ | o
greedy − lemma .onep .fivep _    .(4 + n) n      refl .(4 + l″) ._ | onepfivep | oneponep . _ | o

greedy − lemma .twop .twop _     .n      n       refl l          b (vartype (CoinBag′ twop n l)) |
greedy − lemma .twop .fivep _    .(3 + n) n      refl l          b | twopfivep with view − CoinB
greedy − lemma .twop .fivep _    .(3 + n) n      refl ._         ._ | twopfivep | oneptwop b wit
greedy − lemma .twop .fivep _    .(3 + n) n      refl ._         ._ | twopfivep | oneptwop ._ | o
greedy − lemma .twop .fivep _    .(3 + n) n      refl .(3 + l″) ._ | twopfivep | oneptwop ._ | o

greedy − lemma .twop .fivep _    .(3 + n) n      refl ._         ._ | twopfivep | twoptwop b wit
greedy − lemma .twop .fivep _    .(3 + n) n      refl .(2 + l″) ._ | twopfivep | twoptwop ._ | o

greedy − lemma .twop .fivep _    .(4 + k) .(1 + k) refl .(2 + l″) ._ | twopfivep | twoptwop ._ | t

greedy − lemma .fivep .fivep _   .n      n       refl l          b (vartype (CoinBag′ fivep n l)) |
```

Figure 5.2 Cases of *greedy − lemma*, generated semi-automatically by Agda's interactive case-split mechanism. Shown in the (shaded) interaction points are their goal types, and the types of some pattern variables are shown in subscript beside them.

$$(\_ \leqslant C\_\ d) \ggg\!= \lambda\, e \to \mathit{any} \ggg\!= \lambda\, r \to \mathit{return}\ (\mathit{cons} - \mathit{tag}\ , e\ , r)$$

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 non-trivial to prove the greedy condition by relational calculation. Here we offer instead a brute-force yet conveniently expressed case analysis by dependent pattern matching, which also serves as an example of algebraic ornamentation. Define a new datatype $\mathit{CoinBag}' : \mathit{Coin} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Set}$ by composing two algebraic ornaments on $\mathit{CoinBagD}$ in parallel:

$$\begin{aligned}
&\mathit{CoinBag}'\mathit{OD}\ :\ \mathsf{OrnDesc}\ (\mathsf{outl} \bowtie \mathsf{outl})\ \mathit{pull}\ \mathit{CoinBagD}\\
&\mathit{CoinBag}'\mathit{OD}\ =\ \lceil \mathit{algOrn}\ \mathit{CoinBagD}\ (\mathit{fun}\ \mathit{total} - \mathit{value} - \mathit{alg}) \rceil \otimes\\
&\qquad\qquad\qquad\ \lceil \mathit{algOrn}\ \mathit{CoinBagD}\ (\mathit{fun}\ \mathit{size} - \mathit{alg}) \rceil
\end{aligned}$$

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

By (**??**), (5.2), and $\mathit{fun} - \mathit{preserves} - \mathit{fold}$, $\mathit{CoinBag}'$ is characterised by the isomorphisms

$$\begin{aligned}
\mathit{CoinBag}'\ c\ n\ l\ \cong\ \Sigma\langle\, b : \mathit{CoinBag}\ c\,\rangle&\\
(\mathit{total} - \mathit{value}\ b\ \equiv\ n) \times (\mathit{size}\ b\ \equiv\ l)&
\end{aligned} \tag{5.5}$$

for all $c\ :\ \mathit{Coin}$, $n\ :\ \mathsf{Nat}$, and $l\ :\ \mathsf{Nat}$. Hence a coin bag of type $\mathit{CoinBag}'\ c\ n\ l$ contains l coins that are no larger than c and sum up to n pence. We can give the following types to the two constructors of $\mathit{CoinBag}'$:

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

The greedy condition then essentially reduces to this lemma:

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

That is, given a problem (i.e., a value to be represented by coins), if $c : Coin$ is a choice of decomposition (i.e., the first coin used) no better than $d : Coin$ (recall that we prefer larger denominations), and $b : CoinBag'\ c\ m\ l$ is a solution of size l to the remaining subproblem m resulting from choosing c, then there is a solution to the remaining subproblem n resulting from choosing d whose size l' is no greater than l. We define two *views* McBride and McKinna [2004] — or "customised pattern matching" — to aid the analysis. The first view analyses a proof of $c \leqslant C\ d$ and exhausts all possibilities of c and d,

```
data CoinOrderedView : Coin → Coin → Set where
  oneponep   : CoinOrderedView onep onep
  oneptwop   : CoinOrderedView onep twop
  onepfivep  : CoinOrderedView onep fivep
  twoptwop   : CoinOrderedView twop twop
  twopfivep  : CoinOrderedView twop fivep
  fivepfivep : CoinOrderedView fivep fivep
view − ordered − coin :
  (c d : Coin) → c ⩽ C d → CoinOrderedView c d
```

where the covering function *view − ordered − coin* is written by standard pattern matching on c and d. The second view analyses some $b : CoinBag'\ c\ n\ l$ and exhausts all possibilities of c, n, l, and the first coin in b (if any). The view datatype *CoinBag′View* is shown in Figure 5.1, and the covering function

```
view − CoinBag′ :
  ∀ {c n l} (b : CoinBag′ c n l) → CoinBag′View b
```

is again written by standard pattern matching. Given these two views, *greedy − lemma* can be split into eight cases by first exhausting all possibilities of c and d with *view − ordered − coin* and then analysing the content of b with *view − CoinBag′*. Figure 5.2 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 $b : CoinBag'\ onep\ n\ l$, and we should produce a $CoinBag'\ onep\ n\ l'$ 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 is of type $\mathit{CoinBag}'\ \mathit{onep}\ (1 + n)\ l$, i.e., the coins in the bag are no larger than *onep* and the total value is $1 + n$. The bag must contain *onep* as its first coin; let the rest of the bag be $b\ :\ \mathit{CoinBag}'\ \mathit{onep}\ n\ l''$. 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 *onep* to *twop*, which is done by

  $$\begin{aligned} \mathit{relax}\ :\ &\forall\ \{c\ n\ l\}\ (b\ :\ \mathit{CoinBag}'\ c\ n\ l) \to \\ &\forall\ \{d\} \to c \leqslant C\ d \to \mathit{CoinBag}'\ d\ n\ l \end{aligned}$$

- The remaining goal 6 is the most interesting one: The input bag has type $\mathit{CoinBag}'\ \mathit{twop}\ (3 + n)\ l$, which in this case contains two 2-pence coins, and the rest of the bag is $b\ :\ \mathit{CoinBag}'\ \mathit{twop}\ k\ l''$. 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

  $$\begin{aligned} &\mathit{add} - \mathit{penny}\ : \\ &\quad \forall\ \{c\ n\ l\} \to \mathit{CoinBag}'\ c\ n\ l \to \mathit{CoinBag}'\ c\ (1 + n)\ (1 + l) \end{aligned}$$

  and relax the bound of $\mathit{add} - \mathit{penny}\ b$ from *twop* to *fivep*.

Throughout the proof, Agda is able to keep track of the total value and the size of bags and make deductions, so the case analysis is done with little overhead. The greedy condition can then be discharged by pointwise reasoning, using (5.5) to interface with $\mathit{greedy} - \mathit{lemma}$. We conclude that the Greedy Theorem is applicable, and obtain

$$(\mathit{cata}\ ((\mathit{min}\ Q\ \cdot \Lambda\ (\mathit{fun}\ \mathit{total} - \mathit{value} - \mathit{alg}\ ^{\mathrm{o}}))\ ^{\mathrm{o}}))\ ^{\mathrm{o}} \subseteq \mathit{min} - \mathit{coin} - \mathit{change}$$

We have thus found the algebra

$$R\ =\ (\mathit{min}\ Q\ \cdot \Lambda\ (\mathit{fun}\ \mathit{total} - \mathit{value} - \mathit{alg}\ ^{\mathrm{o}}))\ ^{\mathrm{o}}$$

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:

$$
\begin{aligned}
&\mathit{GreedySolutionOD} \;:\; \mathsf{OrnDesc}\;(\mathit{Coin} \times \mathsf{Nat})\;\mathsf{outl}\;\mathit{CoinBagD}\\
&\mathit{GreedySolutionOD} \;=\; \mathit{algOrn}\;\mathit{CoinBagD}\;R\\
&\mathit{GreedySolution} \;:\; \mathit{Coin} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Set}\\
&\mathit{GreedySolution}\;c\;n \;=\; \mu\;\lfloor \mathit{GreedySolutionOD} \rfloor\;(c\;,n)
\end{aligned}
$$

The two constructors of *GreedySolution* can be given the following types:

$$
\begin{aligned}
&\mathit{gnil} \;:\; \forall\;\{c\;n\} \rightarrow\\
&\qquad \mathit{total}-\mathit{value}-\mathit{alg}\;(\mathit{nil}-\mathit{tag}\;,\blacksquare) \;\equiv\; n \rightarrow\\
&\qquad (\forall\;\mathit{ns} \rightarrow \mathit{total}-\mathit{value}-\mathit{alg}\;\mathit{ns} \;\equiv\; n \rightarrow Q\;\mathit{ns}\;(\mathit{nil}-\mathit{tag}\;,\blacksquare)) \rightarrow\\
&\qquad \mathit{GreedySolution}\;c\;n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{gcons} \;:\\
&\quad \forall\;\{c\;n\} \rightarrow (d \;:\; \mathit{Coin})\;(d \leqslant c \;:\; d \leqslant C\;c) \rightarrow\\
&\quad \forall\;\{n'\} \rightarrow \mathit{total}-\mathit{value}-\mathit{alg}\;(\mathit{cons}-\mathit{tag}\;,d\;,d \leqslant c\;,n') \;\equiv\; n \rightarrow\\
&\quad (\forall\;\mathit{ns} \rightarrow \mathit{total}-\mathit{value}-\mathit{alg}\;\mathit{ns} \;\equiv\; n \rightarrow Q\;\mathit{ns}\;(\mathit{cons}-\mathit{tag}\;,d\;,d \leqslant c\;,n')) \rightarrow\\
&\quad \mathit{GreedySolution}\;d\;n' \rightarrow \mathit{GreedySolution}\;c\;n
\end{aligned}
$$

Before we proceed to construct the internalist program

$$\mathit{greedy} \;:\; (c \;:\; \mathit{Coin})\;(n \;:\; \mathsf{Nat}) \rightarrow \mathit{GreedySolution}\;c\;n$$

let us first simplify the two constructors of *GreedySolution*. Each of the two constructors has two additional proof obligations coming from the algebra R: For *gnil*, since $\mathit{total}-\mathit{value}-\mathit{alg}\;(\mathit{nil}-\mathit{tag},\;\blacksquare)$ reduces to 0, we may just specialise n to 0 and discharge the equality proof obligation. For the second proof obligation, ns is necessarily $(\mathit{nil}-\mathit{tag}\;,\;\blacksquare)$ if $\mathit{total}-\mathit{value}-\mathit{alg}\;\mathit{ns} \;\equiv\; 0$, and indeed Q maps $(\mathit{nil}-\mathit{tag}\;,\;\blacksquare)$ to $(\mathit{nil}-\mathit{tag}\;,\;\blacksquare)$, so the second proof obligation can be discharged as well. We thus obtain a simpler constructor defined using *gnil*:

$$\mathit{gnil}' \;:\; \forall\;\{c\} \rightarrow \mathit{GreedySolution}\;c\;0$$

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

```
UsableCoin : Nat → Coin → Coin → Set
UsableCoin n c d =
  (d ⩽ C c) × (Σ⟨ n′ : Nat ⟩ value d + n′ ≡ n)
```

Now we can define a new constructor using *gcons*:

```
gcons′ :
  ∀ {c} → (d : Coin) → d ⩽ C c →
  ∀ {n′} →
  ((e : Coin) → UsableCoin (value d + n′) c e → e ⩽ C d) →
  GreedySolution d n′ → GreedySolution c (value d + n′)
```

which requires that *d* is the largest usable coin with respect to *c* and *value d* + *n*′. We are thus directed to implement a function *maximum* − *coin* that computes the largest usable coin with respect to any *c* : *Coin* and nonzero *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)
```

which takes some theorem proving but is overall a typical Agda exercise in dealing with natural numbers and ordering. Now we can implement the greedy algorithm as the internalist program

```
greedy : (c : Coin) (n : Nat) → GreedySolution c n
greedy c n = < −rec P f n c
  where
    P : Nat → Set
    P n = (c : Coin) → GreedySolution 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′ (hole () (8)) d)
```

where 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 $\mathit{value}\ d > 0$.

related work: Atkey et al. [2012]

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 2

Robert ATKEY, Patricia JOHANN, and Neil GHANI [2012]. Refining inductive types. *Logical Methods in Computer Science*, 8(2:09). doi:10.2168/LMCS-8(2:9)2012. ↰ pages 22 and 25

Richard BIRD and Oege DE MOOR [1997]. *Algebra of Programming*. Prentice-Hall. ↰ pages 1, 5, 7, and 14

Pierre-Évariste DAGAND and Conor MCBRIDE [2012]. Transporting functions across ornaments. In *International Conference on Functional Programming*, ICFP'12, pages 103–114. ACM. doi:10.1145/2364527.2364544. ↰ page 7

Jeremy GIBBONS, Graham HUTTON, and Thorsten ALTENKIRCH [2001]. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science*, 44(1):146–160. doi:10.1016/S1571-0661(04)80906-X.

Hsiang-Shang KO and Jeremy GIBBONS [2013]. Modularising inductive families. *Progress in Informatics*, 10:65–88. doi:10.2201/NiiPi.2013.10.5.

Conor MCBRIDE [2011]. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*. ↰ page 7

Conor McBride and James McKinna [2004]. The view from the left. *Journal of Functional Programming*, 14(1):69–111. doi:10.1017/S0956796803004829. ↰ page 18

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 1 and 9

Todo list