JFP **19** (5): 545–579, 2009. © 2009 Cambridge University Press
doi:10.1017/S0956796809007345 First published online 23 July 2009 Printed in the United Kingdom

Algebra of programming in Agda: Dependent types for relational program derivation

SHIN-CHENG MU

Institute of Information Science, Academia Sinica, Taiwan
(*e-mail:* scm@iis.sinica.edu.tw)

HSIANG-SHANG KO

Department of Computer Science and Information Engineering, National Taiwan University, Taiwan
(*e-mail:* joshs@mail2000.com.tw)

PATRIK JANSSON

Department of Computer Science and Engineering, Chalmers University of Technology, and University of Gothenburg, Sweden
(*e-mail:* patrikj@chalmers.se)

Abstract

Relational program derivation is the technique of stepwise refining a relational specification to a program by algebraic rules. The program thus obtained is correct by construction. Meanwhile, dependent type theory is rich enough to express various correctness properties to be verified by the type checker. We have developed a library, AoPA (Algebra of Programming in Agda), to encode relational derivations in the dependently typed programming language Agda. A program is coupled with an algebraic derivation whose correctness is guaranteed by the type system. Two non-trivial examples are presented: an optimisation problem and a derivation of quicksort in which well-founded recursion is used to model terminating hylomorphisms in a language with inductive types.

1 Introduction

Program derivation is the technique of successively applying formal rules to a specification until one obtains a program. The program thus obtained is correct by construction. In relational program derivation (Bird & de Moor 1997), specifications are viewed as input–output relations to be stepwise refined by an *algebra of programs*. Meanwhile, type theorists take a complementary approach to program correctness. Modern programming languages deploy advanced type systems that are able to express various correctness properties. This paper aims to show, in the dependently typed language Agda (Norell 2007), how program derivation can be encoded in a type and its proof term. A program and its derivation can thus be written in the same language, and the correctness is guaranteed by the type checker.

The library we have developed, nicknamed AoPA (Algebra of Programming in Agda), is available online (Mu *et al.* 2008b). As a teaser, Figure 1 shows a derivation of a sorting algorithm in progress. The type of *sort-der* is a proposition that there

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

```
sort-der  :  ∃ (λf → ordered? ∘ permute ⊒ fun f)
sort-der  =  (_, (  ordered? ∘ permute
                 ⊒⟨  ∘-mono-r permute-is-fold  ⟩
                    ordered? ∘ foldR combine nil
                 ⊒⟨  foldR-fusion-⊒ ordered? { }0 { }1  ⟩
                    { }2 ))

isort  :  List Val → List Val
isort  =  proj₁ sort-der
```

Fig. 1. A derivation of insertion sort in progress.

exists a function f that, after being lifted to a relation by *fun*, is contained in *ordered*? ∘ *permute*, a relation mapping a list to one of its ordered permutations. (Note that some values in the list may share the same key.) To prove an existential proposition, we provide a pair of a witness and a proof that the witness satisfies the proposition. The witness, once the derivation is complete, can be extracted from the last step of the proof; therefore, we may leave it out as an underscore (_). The first step exploits monotonicity of ∘ and that *permute* can be expressed as a fold. The second step makes use of relational fold fusion (see Section 4.3), but the fusion conditions are not given yet. The shaded areas denote *interaction points* – fragments of (proof) code to be completed – also called meta-variables or just *placeholders* (Magnusson & Nordström 1994). The programmer can query Agda for the expected type and the context of the shaded expression. When the proof is completed, an algorithm *isort* is obtained by extracting the witness of the proposition. (The complete derivation is available online. In Section 6.5 we present a more challenging derivation, quicksort.) It is an executable program that is backed by the type system to meet the specification.

Our work aims to be a cooperation between the *squiggolists*[1] and dependently typed programmers that may benefit both sides:

- This is a case study of using the Curry–Howard isomorphism which the squiggolists may appreciate; specifications of programs are expressed in their types, whose proofs (derivations) are given as programs and checked by the type system. Being able to express derivation *within* the same programming language encourages its use and serves as documentation. In this case study we use dependent types to express and check correctness of the derivations, but the derived functions normally have non-dependent types.
- We modelled a wide range of concepts that often occur in relational program derivation, including relational folds (Backhouse *et al.* 1991), relational division and converse of a function (Mu & Bird 2003). Minimum, for example, is defined using division and intersection, while the *greedy theorem* (Bird & de Moor 1997) is proved using the universal property of minimum. With the theorem we may deal with a number of optimisation problems specified as folds.

[1] 'Squiggol' is a nickname for Bird–Meertens-style program derivation, called so because of the squiggly symbols it uses.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

- In dependently typed programming it is vital to ensure that a program terminates. To deal with unfolds and hylomorphisms (see Section 6), we allow the programmer to model an unfold as the relational converse of a fold but demand a proof of *accessibility* (Nordström 1988) before it is refined to a functional unfold. The connection between accessibility and *inductivity* (Bird & de Moor 1997) is explained.

We originally started to use Agda because of the notation for equality proofs. What started out a as small example grew to a library of around 40 modules and 6,000 lines of code. Interesting future work could be to compare how this library would be expressed in other proof assistants.

For readers not familiar with relational program derivation, a brief overview is given in Section 2, while a quick introduction to a subset of Agda that is relevant to this paper is given in Section 3. After presenting our encoding of relations and their operations in Section 4, we solve an optimisation problem in Section 5, using the greedy theorem. In Section 6 we talk about accessibility, the formal machinery we use to express hylomorphisms, and present a derivation of quicksort as an example. This paper is an extension of the authors' previous work presented at Mathematics of Program Construction (Mu *et al.* 2008a).

Before we go into more technical details, it is probably time to let the readers be aware of some difficulties that those who are familiar with type theory can foresee. We will talk about extensional equality in Section 3.2 and our *ad hoc* approach to get around predicativity in Section 4.1.

2 A quick overview of relational program derivation

One of the most appreciated merits of functional programming is that programs can be manipulated by equational reasoning. Program derivation in the Bird–Meertens style (Bird 1989b) typically start with a specification, as a function, that is obviously correct but not as efficient as one would wish. Various algebraic identities are then applied, in successive steps, to show that the specification equals another functional program that is more efficient. A typical example is the maximum segment sum (Bird 1989a; Gries 1989) problem, whose specification is $max \cdot map\ sum \cdot segs$, where *segs* produces all consecutive segments of the input list of numbers and *map sum* respectively computes their summation, before a maximum is picked by *max*. The specification can be shown, after several steps of transformation, to be equal to another program, whose main computation is performed in a *foldr*, that can be computed in linear time.

There is no mechanical procedure that guarantees to produce efficient programs for all problems in general. The challenge, arguably more contributive than solving a specific problem, is to identify classes of problems that can be manipulated following a certain pattern and to discover and propose useful algebraic properties that play key roles in problem solving.

During the 1990s there was a trend in the program derivation community to move from functions to relations. A specification is given in terms of an input–output

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

relation, which is refined to smaller, more deterministic relations in each step, until we get a function. Relational derivation has some advantages: the specification is often more concise than the corresponding functional specification; optimisation problems can be naturally specified using relations generating all possible solutions (Bird & de Moor 1997; Mu 2008); it is also easier to talk about program inversion (Mu & Bird 2003). The catch, however, is that we now have to reason in terms of inequalities rather than equalities, which is a more challenging task. Much of the groundwork was laid by Backhouse *et al.* (1991) and Backhouse & Hoogendijk (1992). Bird & de Moor (1997) presented program derivation from a category-theoretical point of view, with illustrative examples of problem solving.

A relation R from A to B, denoted by $R : B \leftarrow A$, is usually understood as a subset of the set of pairs $B \times A$. (The 'backward arrow' notation is adopted from Bird & de Moor 1997.) A function f is seen as a special case in which $(b, a) \in f$ and $(b', a) \in f$ implies $b = b'$.

Given a relation $R : B \leftarrow A$, its *converse* $R^{\smile} : A \leftarrow B$ is defined by $(a, b) \in R^{\smile}$ if $(b, a) \in R$. The composition of two relations $R : C \leftarrow B$ and $S : B \leftarrow A$ is defined by: $(c, a) \in R \circ S$ if $\exists b : (c, b) \in R \wedge (b, a) \in S$. Given a relation $R : B \leftarrow A$, its *power transpose* ΛR is a function from A to $\mathbb{P}\, B$ (subsets of B): $\Lambda R\ a = \{\, b \mid (b, a) \in R \,\}$. The relation $\in : A \leftarrow \mathbb{P}\, A$ maps a set to one of its arbitrary members, while $\ni : \mathbb{P}\, A \leftarrow A$ is its converse. The product of two relations, $R \times S$, is defined by $((c, d), (a, b)) \in R \times S$ if $(c, a) \in R$ and $(d, b) \in S$.

In functional programming, the function *foldr* on lists takes a step function of type $A \to B \to B$ and a base case of type B and yields a function $\mathsf{List}\ A \to B$. Its generalisation to a relation, which we denote by *foldR*, remains an important construct. Given an uncurried step relation $R : B \leftarrow (A \times B)$ and a set $s : \mathbb{P}\, B$ recording the base cases, *foldR* R s is a relation having type $B \leftarrow \mathsf{List}\ A$.[2] For example, the relation

$$subseq \quad = \quad foldR\ (cons \cup outr)\ \{[]\},$$

where $cons\ (x, xs) = x :: xs$ and $outr\ (x, xs) = xs$, defines a relation mapping a list to one of its arbitrary subsequences – *cons* keeps the current element, while *outr* drops it.

Relational fold can be defined in terms of functional fold:

$$foldR\ R\ s \quad = \quad \in \circ\ foldr\ \Lambda(R \circ (id \times \in))\ s.$$

To understand the definition, $foldr\ \Lambda(R \circ (id \times \in))\ s$ is a function of type $\mathsf{List}\ A \to \mathbb{P} B$ that collects all the results in a set. The step function $\Lambda(R \circ (id \times \in))$ has type $(A \times \mathbb{P}\, B) \to \mathbb{P}\, B$, where $(id \times \in) : (A \times B) \leftarrow (A \times \mathbb{P}\, B)$ pairs the current element with one of the results from the previous step, before passing the pair to R. It can be proved that *foldR* R s satisfies the universal property

$$foldR\ R\ s = S \quad \Leftrightarrow \quad R \circ (id \times S) = S \circ cons \quad \wedge \quad s = \Lambda S\ [],$$

[2] Isomorphically, the base case can be represented by a relation $B \leftarrow \top$, where $\top$ is the unit type. Furthermore, relational fold, like functional fold, can also be defined for datatypes built from arbitrary regular base functors.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

which in effect states that $foldR\ R\ s$ is the unique fixed point of the monotonic function $\lambda X \rightarrow (R \circ (id \times X) \circ cons^{\smile}) \cup \{(b, []) \mid b \in s\}$. By the Knaster–Tarski theorem (Tarski 1955), $foldR\ R\ s$ is also the least *prefix-point*; therefore we have

$$\begin{array}{l} foldR\ R\ s \subseteq S \quad \Leftarrow \quad R \circ (id \times S) \subseteq S \circ cons \quad \wedge \quad s \subseteq \Lambda S\ [], \\ R \circ (id \times foldR\ R\ s) \subseteq foldR\ R\ s \circ cons \quad \wedge \quad s \subseteq \Lambda(foldR\ R\ s)\ []. \end{array}$$

The first property is called the *induction rule*, while the second the *computation rule* (Backhouse 2002).

If an optimisation problem can be specified by generating all possible solutions using *foldR* or converse of *foldR* before picking the best one, Bird & de Moor (1997) gave a number of conditions under which the specification can be refined to a greedy algorithm, a dynamic programming algorithm or something in between. We will see such an example in Section 5.

3 A crash course in Agda

By 'Agda' we mean Agda version 2, a dependently typed programming language evolved from the theorem prover having the same name. In this section we give a crash course in Agda, focusing on the aspects we need. For detailed documentation, the reader is referred to the main developer's thesis (Norell 2007) and the Agda wiki (Agda Team 2007).

Agda has a Haskell-like syntax extended with a number of additional features. Dependent function types are written $(x : A) \rightarrow B$, where the type expression B may refer to the identifier x, while non-dependent function types are written $A \rightarrow B$. The identity function, for example, can be defined by

$$\begin{array}{lcl} id & : & (A : Set) \rightarrow A \rightarrow A \\ id\ A\ a & = & a, \end{array}$$

where *Set* is the type of *small* types. To apply *id* we should supply both the type and the value parameters, e.g. $id\ \mathbb{N}\ 3$, where $\mathbb{N}$ is the type of natural numbers. Dependently typed programming would be very verbose if we always had to explicitly mention all the parameters. In cases in which some parameters are inferable from the context, the programmer may leave them out, as in $id\ \_\ 3$.

For brevity, Agda supports implicit parameters. In the definition

$$\begin{array}{lcl} id & : & \{A : Set\} \rightarrow A \rightarrow A \\ id\ a & = & a, \end{array}$$

the parameter $\{A : Set\}$ in curly brackets is implicit and need not be mentioned when *id* is called, e.g. *id* 3. Agda tries to infer implicit parameters whenever possible. In case the inference fails, they can be explicitly provided in curly brackets: $id\ \{\mathbb{N}\}\ 3$.

Named parameters in a type signature can be collected in a *telescope*. For example, $\{x : A\} \rightarrow \{y : A\} \rightarrow (z : B) \rightarrow \{w : C\} \rightarrow D$ can be abbreviated to $\{x\ y : A\}(z : B)\{w : C\} \rightarrow D$.

Figure 2 shows some examples of datatype definitions. (We use italic font for identifiers and parameters, sans-serif font for both type and data constructors and

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

```
data List (A : Set) : Set where
  []   :  List A
  _::_ :  A → List A → List A

data ℕ : Set where
  zero :  ℕ
  suc  :  ℕ → ℕ

data _≤_ : ℕ → ℕ → Set where
  ≤-refl  :  {n : ℕ} → n ≤ n
  ≤-step  :  {m n : ℕ} → m ≤ n → m ≤ suc n

_<_     :  ℕ → ℕ → Set
m < n   =  suc m ≤ n
```

Fig. 2. Some examples of datatype definitions.

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B

data Σ (A : Set)(B : A → Set) : Set where
  _,_ : (x : A) → (y : B x) → Σ A B

_×_     :  (A B : Set) → Set
A × B   =  Σ A (λ _ → B)

data ⊥ : Set where

∃   :  {A : Set} (P : A → Set) : Set
∃   =  Σ _

proj₁ : ∀ {A B} → Σ A B → A
proj₁ (x, y) = x

proj₂ : ∀ {A B} →
        (p : Σ A B) → B (proj₁ p)
proj₂ (x, y) = y

record ⊤ : Set where
```

Fig. 3. An encoding of first-order intuitionistic logic in Agda.

boldface font for reserved words.) In Agda's notation for dist-fix definitions, an underscore denotes a location for a parameter. The type constructor List, defining inductive lists, takes a type and yields a type. The parameter $(A : Set)$, written on the left-hand side of the colon, scopes over the entire definition and is an implicit parameter of the constructors _::_ and []. Natural numbers are defined by the type $\mathbb{N}$. The datatype $\_\leqslant\_$ is parameterised not by types but by two *values* of type $\mathbb{N}$. A term having type $m \leqslant n$ represents a proof that m is less than or equal to n. The base case $\leqslant$-refl states that any number is less than or equal to itself, while the inductive case $\leqslant$-step builds a proof of $m \leqslant \mathsf{suc}\ n$ from a proof of $m \leqslant n$. Note that this is merely one of the possible ways to express this proposition.

3.1 First-order intuitionistic logic

In the Curry–Howard isomorphism, types are propositions and terms their proofs. Being able to construct a term of a particular type is to provide a proof of that proposition. Figure 3 shows the encoding of first-order intuitionistic logic in the standard library of Agda (Danielsson *et al.* 2009). Falsity is represented by $\bot$, a type with no constructors and therefore no inhabitants. Truth, on the other hand, can be represented by the record type $\top$, having one unique term – a record with no fields. Disjunction is represented by disjoint sum: a proof of $P \uplus Q$ can be deduced either from a proof of P or a proof of Q. An implication $P \to Q$ is represented as a function taking a proof of P to a proof of Q. We do not introduce new notation for it.

Predicates on type A are represented by $A \to Set$. For example, $(\lambda n \to \mathsf{zero} < n) : \mathbb{N} \to Set$ is a predicate stating that a natural number is positive. Universal quantification of predicate P on type A is encoded as a dependent function type

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

whose elements, given any $x : A$, must produce a proof of $P\ x$. Agda provides a shorthand $\forall\ x \rightarrow P\ x$ in place of $(x : A) \rightarrow P\ x$ when A can be inferred.

The dependent pair Σ is like an ordinary pair, except that the type of the second component may depend on the first component. Functions $proj_1$ and $proj_2$ respectively extract the two components. The product $\_\times\_$, encoding conjunction, is the special case in which the two components are independent. The standard library declares both $\_\times\_$ and $\_,\_$ to be right associative; therefore $(a, (b, c))$ can be abbreviated to (a, b, c).

The existential quantification can also be encoded as a dependent pair: to prove the proposition $\exists\ P$, where P is a predicate on terms of type A, one has to provide, in a pair, a witness $w : A$ and a proof of $P\ w$.

3.2 Identity type

A term of type $x \equiv y$ is a proof that x and y are equal. The datatype $\_\equiv\_$ is defined by

$$\begin{array}{l}\textbf{data}\ \_\equiv\_\ \{A : Set\}(x : A) : A \rightarrow Set\ \textbf{where}\\ \quad \equiv\text{-refl}\ :\ x\ \equiv\ x.\end{array}$$

Agda allows Unicode characters in identifiers; therefore, ≡-refl (without space) is a valid name. For the rest of the paper, we will exploit Unicode characters to give informative names to constructors, arguments and lemmas. For example, if a variable is a proof of $y\ \equiv\ z$, we may name it $y{\equiv}z$ (without space).

The type $\_\equiv\_$ is reflexive by definition. It is also symmetric, meaning that given a term of type $x\ \equiv\ y$, one can construct a term of type $y\ \equiv\ x$:

$$\begin{array}{l}\equiv\text{-}sym\ :\ \{A : Set\}\{x\ y : A\} \rightarrow\ x\ \equiv\ y\ \rightarrow\ y\ \equiv\ x\\ \equiv\text{-}sym\ \{A\}\{x\}\{.x\}\ \equiv\text{-refl}\ =\ \equiv\text{-refl}.\end{array}$$

The implicit arguments, which could be omitted in this case, are given for illustrative purpose. The two occurrences of x appear to imply non-linear pattern matching and run-time equality check. In fact this is not the case. Note that the only constructor ≡-refl is of type $x\ \equiv\ x$. Therefore, if the fourth argument of $\equiv$-*sym* matches ≡-refl and the second argument is x, the third argument could only be x in any well-typed application of $\equiv$-*sym*. This is represented by adding a dot before the second x. A dot pattern corresponds to 'knowledge' rather than 'matching'. It first appeared, with a different notation, in Brady *et al.* (2003).

In dependently typed programming, pattern matching may refine not only the value being inspected but also the types in the context. For $\equiv$-*sym*, now that we have discovered that y could only be x, the return type of $\equiv$-*sym* is instantiated to $x\ \equiv\ x$, for which we can simply return ≡-refl. The algorithm performing matching, unification and context splitting was given by Norell (2007). In general, the combination of pattern matching and inductive families (such as $\_\equiv\_$) is a very powerful one (Dybjer 1994).

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

Some more properties of $\_\equiv\_$ are worth noting. The lemma $\equiv$-*trans* shows that it is transitive:

$$\begin{aligned}&\equiv\text{-}trans\ :\ \{A : Set\}\{x\ y\ z : A\} \to x \equiv y \to y \equiv z \to x \equiv z\\&\equiv\text{-}trans\ \{A\}\{x\}\{.x\}\{z\}\ \equiv\text{-refl}\ x{\equiv}z\ =\ x{\equiv}z.\end{aligned}$$

In the proof of $\equiv$-*trans* we could also replace both $x{\equiv}z$ with $\equiv$-refl. The interactive feature of Agda is helpful for constructing the proof terms. (Agda has an Emacs mode and a command line interpreter interface.) One may, for example, leave out the right-hand side as an interaction point. Agda would prompt the programmer with the expected type of the term to fill in, which also corresponds to the remaining proof obligations. The list of variables in the current context and their types after unification are also available to the programmer.

Furthermore, $\_\equiv\_$ is substitutive – if $x \equiv y$, they are interchangeable in all contexts:

$$\begin{aligned}&\equiv\text{-}subst\ :\ \{A : Set\}(P : A \to Set)\{x\ y : A\} \to x \equiv y \to P\ x \to P\ y\\&\equiv\text{-}subst\ P\ \equiv\text{-refl}\ Px\ =\ Px,\end{aligned}$$

$$\begin{aligned}&\equiv\text{-}cong\ :\ \{A\ B : Set\}(f : A \to B)\{x\ y : A\} \to x \equiv y \to f\ x \equiv f\ y\\&\equiv\text{-}cong\ f\ \equiv\text{-refl}\ =\ \equiv\text{-refl}.\end{aligned}$$

However, $\_\equiv\_$ is not extensional. In Agda, equality of terms is checked by expanding them to normal forms. We therefore have problem comparing higher-order values: $sum \cdot map\ sum$ and $sum \cdot concat$, for example, while defining the same function summing up a list of lists of numbers are not 'equal' under $\_\equiv\_$. One may define extensional equality for (non-dependent) functions on first-order values:

$$\begin{aligned}&\dot{=}\_ : \{A\ B : Set\} \to (A \to B) \to (A \to B) \to Set\\&f \dot{=} g\ =\ \forall\ x \to f\ x \equiv g\ x.\end{aligned}$$

However, $\dot{=}\_$ is not substitutive. Congruence of $\dot{=}\_$ has to be proved for each context.

As we will see later, refinement in preorder reasoning inevitably involves replacing terms in provably monotonic contexts. This is a significant overhead, but given that this overhead is incurred anyway, not having extensional equality is no extra trouble.

3.3 Preorder reasoning

To prove a proposition $e_1 \equiv e_2$ is to construct a term having that type. One *can* do that, using the operators defined in the previous section. But it can be very tedious, when the expressions involved get complicated. Luckily, for any binary relation $\_\sim\_$ that is reflexive and transitive (that is for which one can construct terms $\sim$-*refl* and $\sim$-*trans* having the types as described in the previous section), we can derive a set of combinators, shown in Figure 4, which allows one to construct a term of type $e_1 \sim e_n$ in algebraic style. These combinators were implemented in Agda by Norell (2007) and improved by Danielsson in the standard library of Agda (Danielsson *et al.* 2009). Augustsson (1999) has proposed a similar syntax for equality reasoning, with automatic inference of congruences.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

```
infixr  2  _∼⟨_⟩_
infix   2  _∼□

_∼⟨_⟩_ : {A : Set}(x : A){y z : A} →  x ∼ y  →  y ∼ z  →  x ∼ z
x  ∼⟨  x∼y  ⟩  y∼z  =  ∼-trans  x∼y  y∼z

_∼□ : {A : Set}(x : A) →  x ∼ x
x ∼□  =  ∼-refl
```

Fig. 4. Combinators for preorder reasoning.

To understand the definitions, note that $\_\sim\langle\_\rangle\_$, a dist-fix function taking three explicit parameters, associates to the right. Therefore, the algebraic proof

$$\begin{array}{ll} & e_1 \\ \sim\langle & reason_1 \quad \rangle \\ & \vdots \\ & e_{n-1} \\ \sim\langle & reason_{n-1} \quad \rangle \\ & e_n \\ \sim\Box & \end{array}$$

should be bracketed as $e_1 \sim\langle reason_1\rangle \ldots (e_{n-1} \sim\langle reason_{n-1}\rangle\ (e_n\sim\Box))$. Each occurrence of $\_\sim\langle\_\rangle\_$ takes three arguments – e_i on the left, $reason_i$ (a proof that $e_i \sim e_{i+1}$) in the angle brackets and a proof of $e_{i+1} \sim e_n$ on the right-hand side – and produces a proof of $e_i \sim e_n$ using $\sim$-*trans*. As the base case, $\sim\Box$ takes the value e_n and returns a term of type $e_n \sim e_n$.

We have seen in the previous section that $\_\equiv\_$ is reflexive and transitive. For another useful example, we may define implication as a relation:

$$\begin{array}{ll} \_\Rightarrow\_ \ :\ Set \to Set \to Set & \_\Leftarrow\_ \ :\ Set \to Set \to Set \\ P \Rightarrow Q \ =\ P \to Q, & P \Leftarrow Q \ =\ Q \Rightarrow P. \end{array}$$

Reflexivity and transitivity of $\_\Leftarrow\_$, for example, can be simply given by $\Leftarrow$-*refl* $= id$ and $\Leftarrow$-*trans* $= \_\cdot\_$, where $\_\cdot\_$ is function composition. Therefore, they induce a pair of operators $\_\Leftarrow\langle\_\rangle\_$ and $\Leftarrow\Box$ for logical reasoning.

There is a slight complication, however. Agda maintains a hierarchy of universes, where *Set*, the type of small types, is in sort *Set* 1. While instantiating $\_\sim\_$ in Figure 4 to $\_\Leftarrow\_ : Set \to Set \to Set$, one would notice that the type $A : Set$ itself cannot be instantiated to *Set*, which is in *Set* 1. Currently, we resolve the problem simply by using different module generators for different universes. More on this will be given in Section 4.1.

3.4 Functional derivation

The ingredients we have prepared so far already allow us to perform some functional program derivation. Since $\_\dot{=}\_$ can be shown to be reflexive and transitive, it also induces its preorder reasoning operators. Figure 5 shows a proof of the universal

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

```
foldr                : {A B : Set} → (A → B → B) → B → List A → B
foldr f e []         = e
foldr f e (x :: xs)  = f x (foldr f e xs)

foldr-universal : ∀ {A B} (h : List A → B) f e →
  (h [] ≡ e) → (∀ x xs → h (x :: xs) ≡ f x (h xs)) → h ≐ foldr f e
foldr-universal h f e base step []        = base
foldr-universal h f e base step (x :: xs) =
  h (x :: xs)
  ≡⟨ step x xs ⟩
  f x (h xs)
  ≡⟨ ≡-cong (f x) (foldr-universal h f e base step xs) ⟩
  f x (foldr f e xs)
  ≡⟨ ≡-refl ⟩
  foldr f e (x :: xs)
  ≡□
```

Fig. 5. Proving the universal property for *foldr*.

```
scanr-der : {A B : Set} → (f : A → B → B) → (e : B) →
              ∃ (λprog → map⁺ (foldr f e) · tails ≐ prog)
scanr-der f e = (_, (  map⁺ (foldr f e) · tails
                    ≐⟨ foldr-fusion (map⁺ (foldr f e)) (push-map-til f) ⟩
                      foldr (sc f) [ e ]⁺
                    ≐□))
  where sc : {A B : Set} → (A → B → B) → A → List⁺ B → List⁺ B
        sc f a [ b ]⁺       = f a b ::⁺ [ b ]⁺
        sc f a (b ::⁺ bs)   = f a b ::⁺ b ::⁺ bs
        push-map-til : {A B : Set} → (f : A → B → B) → {e : B} → (a : A) →
              map⁺ (foldr f e) · til a ≐ sc f a · map⁺ (foldr f e)
        push-map-til f a [ xs ]⁺       = ≡-refl
        push-map-til f a (xs ::⁺ xss)  = ≡-refl
```

Fig. 6. Derivation of *scanr*. The constructors $\_::^+\_$ and $[\_]^+$ build non-empty lists, while $tails = foldr\ til\ [[]]^+$, where $til\ a\ [\,xs\,]^+ = (a :: xs) ::^+ [\,xs\,]^+$; $til\ a\ (xs\ ::^+ xss) = (a :: xs)\ ::^+ xs\ ::^+ xss$.

property of *foldr*. The steps using ≡-refl are simple equivalences which Agda can prove by expanding the definitions. The inductive hypothesis is established by a recursive call to *foldr-universal*. Agda ensures that proofs by induction are well founded. With the universal property we can prove the *foldr-fusion* theorem:

```
foldr-fusion : ∀ {A B C} (h : B → C) {f : A → B → B} {g : A → C → C} →
    {e : B} → (∀ x y → h (f x y) ≡ g x (h y)) → h · foldr f e ≐ foldr g (h e)
foldr-fusion h {f} {g} e fuse =
    foldr-universal (h · foldr f e) g (h e) ≡-refl (λx xs → fuse x (foldr f e xs)).
```

Figure 6 derives *scanr* from its specification $map^+\ (foldr\ f\ e) \cdot tails$, where map^+ is the map function defined for List^+, the type of non-empty lists. The *foldr-fusion* theorem is used to transform the specification to a fold. The derived program can be extracted by $scanr = proj_1\ scanr\text{-}der$, while $scanr\text{-}pf = proj_2\ scanr\text{-}der$ is a proof that can be used elsewhere. Note that the first component of the pair (the witness)

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

is left implicit. Agda is able to infer the witness because it is syntactically presented in the derivation as $foldr\ (sc\ f)\ [\ e\ ]^{+}$.

We have reproduced a complete derivation for the maximum segment sum problem. The derivation proceeds in the standard manner (Bird 1989a), transforming the specification to $max \cdot map\ (foldr\ \_\otimes\_\ 0) \cdot tails$ for some $\_\otimes\_$ and exploiting *scanr-pf* to convert it to a *scanr*. The main derivation is about 230 lines long, plus 400 lines of library code proving properties about lists and 100 lines for properties about integers. The code is available online (Mu *et al.* 2008b).

The interactive interface of Agda proved to be very useful. One could progress the derivation line by line, leaving out the unfinished part as an interaction point. One may also type in the desired next step but leave the 'reason' part blank and let Agda derive the type of the lemma needed.

4 Modelling relations

Many definitions in Bird & de Moor (1997) are given in terms of *universal properties*, which are also useful rules for calculation. In this work, however, we give (a constructive variant of) set-theoretical definitions and prove the universal properties afterwards.

4.1 Sets and relations

A possibly infinite but decidable subset of A could be represented by its membership function of type $A \to Bool$. With this representation, however, some operations to be introduced later, such as relational composition, cannot be implemented when the domain is infinite. With dependent types, we can represent the membership judgement at type level:

$$\begin{aligned} \mathbb{P}\ &: Set \to Set\,1 \\ \mathbb{P}\,A\ &= A \to Set. \end{aligned}$$

A set $s : \mathbb{P}\,A$ is a function mapping $a : A$ to a type, which encodes a logic formula determining its membership. The formula need not be decidable in general, but for the programs we derive it will be. As mentioned before, Agda maintains a hierarchy of universes. Set denotes the universe of small types; $Set\,1$ denotes the universe of Set and all types declared as being in $Set\,1$; and so on. Since $s : \mathbb{P}\,A$ is a function yielding a Set, $\mathbb{P}\,A$ is in the universe $Set\,1$.

Set union, intersection and inclusion, for example, are naturally encoded by disjunction, conjunction and implication, as shown in Figure 7. The relation $\_\subseteq\_$ can be shown to be reflexive and transitive. Note how, in the proof of $\subseteq$-*trans*, a proof of $r \subseteq s$ is applied to a and $a \in r$ to produce a proof of $a \in s$. The function *singleton* creates singleton sets:

$$\begin{aligned} singleton\ &: \{A : Set\} \to A \to \mathbb{P}\,A \\ singleton\ a\ &= \lambda a' \to a \equiv a'. \end{aligned}$$

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

$$\begin{array}{ll}
\cup : \{A : Set\} \to \mathbb{P}A \to \mathbb{P}A \to \mathbb{P}A & _\subseteq_ : \{A : Set\} \to \mathbb{P}A \to \mathbb{P}A \to Set \\
(r \cup s)\ a = r\ a \uplus s\ a & r \subseteq s = \forall\ a \to r\ a \to s\ a \\
\\
\cap : \{A : Set\} \to \mathbb{P}A \to \mathbb{P}A \to \mathbb{P}A & _\supseteq_ : \{A : Set\} \to \mathbb{P}A \to \mathbb{P}A \to Set \\
(r \cap s)\ a = r\ a \times s\ a & r \supseteq s = s \subseteq r \\
\\
\subseteq\text{-}refl : \{A : Set\} \to \{r : \mathbb{P}A\} \to r \subseteq r & \subseteq\text{-}trans : \{A : Set\} \to \{r\ s\ t : \mathbb{P}A\} \to \\
\subseteq\text{-}refl\ a\ a{\in}r = a{\in}r & \qquad r \subseteq s \to s \subseteq t \to r \subseteq t \\
 & \subseteq\text{-}trans\ r{\subseteq}s\ s{\subseteq}t\ a\ a{\in}r = s{\subseteq}t\ a\ (r{\subseteq}s\ a\ a{\in}r)
\end{array}$$

Fig. 7. Set union, intersection and inclusion.

However, recall the discussion in Section 3.2, and note that the definition above is intended to handle only sets of first-order values. Currently it is sufficient for all program derivation problems we have dealt with.

A relation $B \leftarrow A$, seen as a set of pairs, could be represented as $\mathbb{P}(B \times A) = (B \times A) \to Set$. However, we find the following 'curried' representation more convenient:

$$\begin{array}{l}
\leftarrow : Set \to Set \to Set\,1 \\
B \leftarrow A = B \to A \to Set.
\end{array}$$

Therefore, given $R : B \leftarrow A$, the proposition that R maps a to b is represented by $R\ b\ a$. The order of arguments is picked so that when R is taken to be $\_<\_$, the 'output' b is the smaller one, which will come in handy in Section 6.

A function on first-order values can be converted to a relation:

$$\begin{array}{l}
fun : \{A\ B : Set\} \to (A \to B) \to (B \leftarrow A) \\
fun\ f\ b\ a = f\ a \equiv b.
\end{array}$$

The identity relation, for example, is denoted $id_R : \{A : Set\} \to (A \leftarrow A)$ and defined by $id_R = fun\ id$. On the other hand, the Λ operator converts a relation to a set-valued function:

$$\begin{array}{l}
\Lambda : \{A\ B : Set\} \to (B \leftarrow A) \to (A \to \mathbb{P}\,B) \\
\Lambda\ R\ a = \lambda b \to R\ b\ a.
\end{array}$$

Relational composition is defined by

$$\begin{array}{l}
\circ : \{A\ B\ C : Set\} \to (C \leftarrow B) \to (B \leftarrow A) \to (C \leftarrow A) \\
(R \circ S)\ c\ a = \exists\ (\lambda b \to (R\ c\ b \times S\ b\ a)).
\end{array}$$

Definitions of some more operators, including $\_^{\smile}$ for relational converse and $\_\times\_$, the product functor, are given in Figure 8.

Complication arises when we try to represent $\in$ and $\ni$. Recall that $\in$ maps $\mathbb{P}\,A$ to A. However, the arguments to $\_\leftarrow\_$ must be in Set, while $\mathbb{P}\,A$ is in $Set\,1$. We may declare types of arrows whose inputs or outputs are $Set\,1$-sorted:

$$\begin{array}{ll}
_\leftarrow_1_ : Set \to Set\,1 \to Set\,1 & _{}_1\!\leftarrow_ : Set\,1 \to Set \to Set\,1 \\
B \leftarrow_1 A = A \to B \to Set, & B\ {}_1\!\leftarrow A = A \to B \to Set.
\end{array}$$

But that means we need several versions of all the relational operators that differ only in their types. Such inconvenience may be resolved if Agda introduces *universe polymorphism* (Harper & Pollack 1991), a feature that was being discussed in the

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

```
_˘ : {A B : Set} → (A ← B) → (B ← A)
(R ˘) a b = R b a

_⨯_ : {A B C D : Set} → (B ← A) → (D ← C) → ((B × D) ← (A × C))
(R ⨯ S) (b, d) (a, c) = R b a × S d c

∈ : {A : Set} → (A ←₁ ℙ A)          ∋ : {A : Set} → (ℙ A ₁← A)
∈ a pa = pa a                       ∋ pa a = pa a

_₁∘_ : {A B C : Set} → (C ←₁ ℙ B) → (A → ℙ B) → (C ← A)
(R ₁∘ S) c a = R c (S a)
```

Fig. 8. Some relational operators. Distinguish between the type of pairs (_×_) and its functor action on relations (_⨯_).

Agda community at the time of writing. In the actual code we painstakingly defined all the variations we need. For presentation in this paper, however, we pretend that our relational operators are polymorphic with respect to universes.

We would still run into trouble if we try to compose a relation $C \leftarrow_1 B$ with $B \ {}_1\!\leftarrow A$, which should ideally yield a relation of type $C \leftarrow A = C \rightarrow A \rightarrow Set$. The proposition $\exists\ \{B\}\ (\lambda b \rightarrow (R\ c\ b \times S\ b\ a))$, however, quantifies over B, which is in $Set\,1$, and therefore the proposition cannot be in Set. Luckily, for our purpose, the only type in $Set\,1$ we need is the powerset, and we can consider this special case only.[3] The operator $\_{}_1\circ\_$, defined in Figure 8, composes a relation $C \leftarrow_1 \mathbb{P}\,B$ with a function $A \rightarrow \mathbb{P}\,B$ to yield a relation $C \leftarrow A$. Another option would be to construct a user-defined universe (Dybjer & Setzer 1999), an alternative we are yet to explore.

4.2 Inclusion and monotonicity

A relation S can be refined to R if every possible outcome of R is a legitimate outcome of S. We represent the refinement relation by

$$\begin{aligned} &\_\sqsubseteq\_ : \{A\ B : Set\} \rightarrow (B \leftarrow A) \rightarrow (B \leftarrow A) \rightarrow Set \\ &R \sqsubseteq S = \forall\ b\ a \rightarrow R\ b\ a \rightarrow S\ b\ a, \end{aligned}$$

Conversely, $R \sqsupseteq S = S \sqsubseteq R$. It is shown in Figure 9 that $\_\sqsubseteq\_$ is reflexive and transitive, and therefore, so is $\_\sqsupseteq\_$. We can thus use them for preorder reasoning.

It is also shown in Figure 9 that composition is associative and monotonic. The proof for associativity follows from associativity of the existential quantifier. While the proof terms may look tedious, they can be easily constructed with the help of Agda's interactive mode. These kinds of lemmas, and their uses, are examples of what a Coq (Coq Development Team 2006) style tactic or a first-order logic plug-in could automate.

[3] Alternatively, one may turn off the universe check in the current implementation of Agda by a compiler flag `--type-in-type`, at the expense of allowing Girard's paradox to be encoded. We did not take this route because once we do so, there may be no easy way back.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

$$
\begin{aligned}
&\sqsubseteq\text{-}refl \;:\; \{A\ B : Set\}\{R : B \leftarrow A\} \to R \sqsubseteq R\\
&\sqsubseteq\text{-}refl \;\_ \;=\; \subseteq\text{-}refl
\end{aligned}
$$

$$
\begin{aligned}
&\sqsubseteq\text{-}trans \;:\; \{A\ B : Set\}\{R\ S\ T : B \leftarrow A\} \to R \sqsubseteq S \to S \sqsubseteq T \to R \sqsubseteq T\\
&\sqsubseteq\text{-}trans\ \ R{\sqsubseteq}S\ \ S{\sqsubseteq}T\ \ b \;=\; \subseteq\text{-}trans\ \ (R{\sqsubseteq}S\ \ b)\ \ (S{\sqsubseteq}T\ \ b)
\end{aligned}
$$

$$
\begin{aligned}
&\circ\text{-}assocl \;:\; \{A\ B\ C\ D : Set\}\ \{R : D \leftarrow C\}\ \{S : C \leftarrow B\}\ \{T : B \leftarrow A\} \to\\
&\qquad\qquad R \circ (S \circ T) \sqsubseteq (R \circ S) \circ T\\
&\circ\text{-}assocl\ \ d\ a\ (c, dRc, (b, cSb, bTa)) \;=\; (b, (c, dRc, cSb), bTa)
\end{aligned}
$$

$$
\begin{aligned}
&\circ\text{-}assocr \;:\; \{A\ B\ C\ D : Set\}\ \{R : D \leftarrow C\}\ \{S : C \leftarrow B\}\ \{T : B \leftarrow A\} \to\\
&\qquad\qquad (R \circ S) \circ T \sqsubseteq R \circ (S \circ T)\\
&\circ\text{-}assocr\ \ d\ a\ (b, (c, dRc, cSb), bTa) \;=\; (c, dRc, (b, cSb, bTa))
\end{aligned}
$$

$$
\begin{aligned}
&\circ\text{-}mono\text{-}l : \{A\ B\ C : Set\}\ \{T : B \leftarrow A\}\ \{R\ S : C \leftarrow B\} \to R \sqsubseteq S \to R \circ T \sqsubseteq S \circ T\\
&\circ\text{-}mono\text{-}l\ \ R{\sqsubseteq}S\ \ c\ \ a\ \ (b, cRb, bTa) \;=\; (b, R{\sqsubseteq}S\ c\ b\ cRb, bTa)
\end{aligned}
$$

$$
\begin{aligned}
&\circ\text{-}mono\text{-}r : \{A\ B\ C : Set\}\ \{T : C \leftarrow B\}\ \{R\ S : B \leftarrow A\} \to R \sqsubseteq S \to T \circ R \sqsubseteq T \circ S\\
&\circ\text{-}mono\text{-}r\ \ R{\sqsubseteq}S\ \ c\ \ a\ \ (b, cRb, bRa) \;=\; (b, cTb, R{\sqsubseteq}S\ b\ a\ bRa)
\end{aligned}
$$

Fig. 9. Some properties of relations.

Another lemma often used without being said is that we can introduce id_R to the right of any relation:

$$
\begin{aligned}
&id\text{-}intro\text{-}r \;:\; \{A\ B\ : Set\}\{R : B \leftarrow A\} \to R \sqsupseteq R \circ id_R\\
&id\text{-}intro\text{-}r\ b\ a\ (.a, bRa, \equiv\text{-refl}) \;=\; bRa.
\end{aligned}
$$

The arguments a and b are respectively the input and output of $R \circ id_R$, while the third argument is a proof that there exists some value connecting R and id_R. Due to the presence of $\equiv$-refl, the value could only be a. The dot pattern was introduced in Section 3.2.

4.3 Relational fold

We can now define relational fold. Using

$$
\begin{aligned}
&foldr_1 \;:\; \{A : Set\} \to \{PB\ : Set\,1\} \to ((A \times B) \to PB) \to PB \to \mathsf{List}\ A \to PB\\
&foldr_1\ f\ e\ \ [\,] \qquad\quad = \; e\\
&foldr_1\ f\ e\ (x :: xs) \;=\; f\ (x, foldr_1\ f\ e\ xs)
\end{aligned}
$$

which is the $Set\,1$-kinded variation of $foldr$, the relational fold can be defined in terms of $foldr_1$:

$$
\begin{aligned}
&foldR \;:\; \{A\ B\ : Set\} \to (B \leftarrow (A \times B)) \to \mathbb{P}\,B \to (B \leftarrow \mathsf{List}\ A)\\
&foldR\ R\ s \;=\; \in_1 \circ foldr_1\ (\Lambda(R \circ (id_R \times \in)))\ s.
\end{aligned}
$$

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

Define the relational version of the list constructors $cons = fun\ (uncurry\ \_::\_)$ and $nil = singleton\ []$. The induction and computation rules are as follows:

$$\begin{aligned}
&foldR\text{-}induction\text{-}\sqsupseteq\ :\ \{A\ B\ :Set\}\ (S\ :B \leftarrow \mathsf{List}\ A)\ (R\ :B \leftarrow (A \times B))\ (e\ :\mathbb{P}\ B) \rightarrow \\
&\qquad (R \circ (id_R \between S) \sqsubseteq S \circ cons) \times (e \subseteq \Lambda(S \circ \in)\ nil) \rightarrow foldR\ R\ e \sqsubseteq S \\
&foldR\text{-}computation\text{-}\sqsupseteq\ :\ \{A\ B\ :Set\}\ (R\ :B \leftarrow (A \times B))\ (e\ :\mathbb{P}\ B) \rightarrow \\
&\qquad (R \circ (id_R \between foldR\ R\ e) \sqsubseteq foldR\ R\ e \circ cons) \times (e \subseteq foldR\ R\ e\ []).
\end{aligned}$$

The proof, omitted here but available online (Mu *et al.* 2008b), proceeds by converting both sides to functional folds and using induction. From the induction rule, the fusion theorem follows:

$$\begin{aligned}
&foldR\text{-}fusion\text{-}\sqsupseteq\ :\ \{A\ B\ C\ :Set\}\ (R\ :C \leftarrow B)\ \{S\ :B \leftarrow (A \times B)\} \rightarrow \\
&\quad \{T\ :C \leftarrow (A \times C)\}\{u\ :\mathbb{P}\ B\}\{v\ :\mathbb{P}\ C\} \rightarrow \\
&\qquad R \circ S \sqsupseteq T \circ (id_R \between R)\ \rightarrow\ \Lambda(R \circ \in)\ u \supseteq v\ \rightarrow\ R \circ foldR\ S\ u \sqsupseteq foldR\ T\ v.
\end{aligned}$$

To use fold fusion, however, there has to be a fold to start with. The following lemma shows that id_R, when instantiated to lists, is a fold (the type argument $\mathsf{List}\ A$ to id_R is there to aid the type checker):

$$\begin{aligned}
id_R \sqsupseteq foldR\ &:\ \{A : Set\} \rightarrow id_R\ \{\mathsf{List}\ A\}\ \sqsupseteq\ foldR\ cons\ nil \\
id_R \sqsupseteq foldR\ &=\ foldR\text{-}induction\text{-}\sqsupseteq\ id_R\ cons\ nil\ (idstep, idbase).
\end{aligned}$$

We can therefore introduce an id_R wherever we want, turn it into a fold and perform fusion. The proof makes use of the induction rule, where the two premises are trivial to prove:

$$\begin{aligned}
idstep\ &:\ id_R \circ\ cons\ \sqsupseteq\ cons \circ (id_R \between id_R) \\
idbase\ &:\ \Lambda(id_R \circ \in)\ nil \supseteq nil.
\end{aligned}$$

4.4 Relational division

Given relations $R\ :B \leftarrow A$ and $S\ :C \leftarrow A$, the right division $R/S\ :B \leftarrow C$ is characterised by the following universal property:

$$X \circ S\ \sqsubseteq\ R\ \Leftrightarrow\ X\ \sqsubseteq\ R/S.$$

That is R/S is the largest relation such that $R/S \circ S \sqsubseteq R$. Read set theoretically, it says that $(b, c) \in R/S$ if and only if for all a, $(c, a) \in S$, (b, a) is in R. That translates to the Agda definition

$$\begin{aligned}
_/_\ &:\ \{A\ B\ C\ :Set\} \rightarrow (B \leftarrow A) \rightarrow (C \leftarrow A) \rightarrow (B \leftarrow C) \\
(R/S)\ b\ c\ &=\ \forall\ a \rightarrow S\ c\ a \rightarrow R\ b\ a,
\end{aligned}$$

given that we may prove the universal property. The left division, on the other hand, can be given by

$$\begin{aligned}
_\backslash_\ &:\ \{A\ B\ C\ :Set\} \rightarrow (B \leftarrow A) \rightarrow (B \leftarrow C) \rightarrow (A \leftarrow C) \\
R\backslash S\ &=\ (S^{\smile}/R^{\smile})^{\smile}.
\end{aligned}$$

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

```
min-universal-⇐ : {A B : Set} (⊴ : A ← A) (S : A ← B) (X : A ← B) →
                  (X ⊑ S) × (X ∘ S˘ ⊑ ⊴) → X ⊑ min ⊴ ₁∘ ΛS
min-universal-⇐ ⊴ S X =
    X ⊑ min ⊴ ₁∘ ΛS
1 : ⇐⟨ ⇐-refl ⟩
    X ⊑ (∈ ⊓ (⊴/∋)) ₁∘ ΛS
2 : ⇐⟨ ⊒-trans (⊓-Λ-distr-⊒ ∈ (⊴/∋) S) ⟩
    X ⊑ (∈ ₁∘ ΛS) ⊓ ((⊴/∋) ₁∘ ΛS)
3 : ⇐⟨ ⊓-universal-⇐ ⟩
    (X ⊑ S) × (X ⊑ (⊴/∋) ₁∘ ΛS)
4 : ⇐⟨ ⇐-refl ⟩
    (X ⊑ S) × (X ⊑ ⊴/S˘)
5 : ⇐⟨ map-× id /-universal-⇒ ⟩
    (X ⊑ S) × (X ∘ S˘ ⊑ ⊴)
  ⇐□
```

Fig. 10. Proving the universal property of *min*.

4.5 Minimum and maximum

Let $\_\trianglelefteq\_ : A \leftarrow A$ be a relation representing an ordering. For brevity we sometimes omit the underlines and just write $\trianglelefteq$. The relation $min\ \trianglelefteq$ maps a set of A-values to one of its minimum element with respect to $\trianglelefteq$. Bird & de Moor (1997) used *min* to model optimisation problems. Relational intersection can be defined as

$$\begin{array}{lcl} \_\sqcap\_ & : & \{A\ B : Set\} \to (B \leftarrow A) \to (B \leftarrow A) \to (B \leftarrow A) \\ (R \sqcap S)\ b\ a & = & R\ b\ a \times S\ b\ a, \end{array}$$

which satisfies the universal property $R \sqsubseteq (S \sqcap T) \Leftrightarrow (R \sqsubseteq S) \times (R \sqsubseteq T)$. The relation $min\ \trianglelefteq$ is then defined by

$$\begin{array}{lcl} min & : & \{A : Set\} \to (A \leftarrow A) \to (A \leftarrow \mathbb{P}\ A) \\ min\ \trianglelefteq & = & \in\ \sqcap\ (\trianglelefteq/\ni). \end{array}$$

To understand the definition, assume that $min\ \trianglelefteq$ maps a set s to a. The left-hand side of $\_\sqcap\_$ ensures that a is an element of s, while the division on the right-hand side states that for any $a' \in s$, we must have $a \trianglelefteq a'$. Dually, to pick a maximum element in a set, we define $max\ \trianglelefteq = min\ (\trianglelefteq\breve{})$.

The relation $min\ R$ also satisfies a universal property:

$$X \sqsubseteq min\ \trianglelefteq\ {}_1\circ\ \Lambda S \quad \Leftrightarrow \quad (X \sqsubseteq S) \times (X \circ S\breve{} \sqsubseteq \trianglelefteq).$$

Figure 10 shows a proof of the 'if' direction. We actually need *Set* 1-kinded operators in some of the steps, but, as stated before, we omit the detail for presentation. Apart from that, the proof is almost identical to a hand-written proof. The lemma ⊓-Λ-*distr*-⊒ in step 2 allows one to move a set-valued function out of intersection: $(R\ {}_1\circ\ \Lambda T) \sqcap (S\ {}_1\circ\ \Lambda T) \sqsubseteq (R \sqcap S)\ {}_1\circ\ \Lambda T$. Step 2 shall prove that $X \sqsubseteq Y \Leftarrow X \sqsubseteq Z$, given $Z \sqsubseteq Y$, which is exactly what ⊒-*trans* establishes. Step 3 uses the universal property of $\_\sqcap\_$ to split the premise into a conjunction. Step 4 shifts ΛS into the quotient of the division, transforming $(\trianglelefteq/\ni)\ {}_1\circ\ \Lambda S$ into $\trianglelefteq/S\breve{}$. Agda is able to see that they are equal by expanding the definition of $\_/\_$; therefore we may simply put

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

$$
\begin{aligned}
&Act \;:\; Set\\
&Act \;=\; \Sigma\ (\mathbb{N}\times\mathbb{N})\ (\lambda p \to proj_1\ p < proj_2\ p).
\end{aligned}
$$

$$
\begin{aligned}
&disjoint \;:\; Act \leftarrow Act\\
&disjoint\ a\ x \;=\; finish\ x \leq start\ a\ \uplus\ finish\ a \leq start\ x
\end{aligned}
$$

$$
\begin{aligned}
&compatible \;:\; \mathbb{P}\,(Act \times \mathsf{List}\ Act)\\
&compatible\ (a, xs) \;=\; all\ (disjoint\ a)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&mutex \;:\; \mathsf{List}\ Act \leftarrow \mathsf{List}\ Act\\
&mutex \;=\; check\ (compatible\,¿)
\end{aligned}
$$

$$
\begin{aligned}
&lessfin \;:\; Act \leftarrow Act\\
&lessfin\ a\ x \;=\; finish\ x \leq finish\ a
\end{aligned}
$$

$$
\begin{aligned}
&fin\text{-}ubound \;:\; \mathbb{P}\,(Act \times \mathsf{List}\ Act)\\
&fin\text{-}ubound\ (a, xs) \;=\; all\ (lessfin\ a)\ xs
\end{aligned}
$$

$$
\begin{aligned}
&fin\text{-}ordered \;:\; \mathsf{List}\ Act \leftarrow \mathsf{List}\ Act\\
&fin\text{-}ordered \;=\; check\ (fin\text{-}ubound\,¿)
\end{aligned}
$$

$$
\begin{aligned}
&subseq \;:\; \{A : Set\} \to (\mathsf{List}\ A \leftarrow \mathsf{List}\ A)\\
&subseq \;=\; foldR\ (outr \sqcup cons)\ nil
\end{aligned}
$$

$$
\begin{aligned}
&\_\leq_l\_ \;:\; \{A : Set\} \to (\mathsf{List}\ A \leftarrow \mathsf{List}\ A)\\
&xs \leq_l ys \;=\; length\ xs \leq length\ ys
\end{aligned}
$$

$$
\begin{aligned}
&act\text{-}sel\text{-}spec \;:\; \mathsf{List}\ Act \leftarrow \mathsf{List}\ Act\\
&act\text{-}sel\text{-}spec \;=\; (max\ \leq_l 1 \circ \Lambda(mutex \circ subseq)) \circ fin\text{-}ordered
\end{aligned}
$$

Fig. 11. Specifying the activity-selection problem.

⇐-*refl* as the reason. We can then use the universal property of $\_/\_$ in step 5, applied to the right component of the conjunction by the combinator *map*-×.

5 Example: The activity-selection problem

We are finally in a position to present an example. In this section we pick an optimisation problem, with a fairly simple algorithm, that is small enough to fit within this paper yet demonstrates the use of the greedy theorem (Section 5.2) – with a little twist, however.

Given a list of activities, each labelled with its start time and finish time, the *activity-selection problem* (Cormen *et al.* 2001) is to choose as many non-overlapping activities as possible. When the list of activities is sorted by their finish time, there is a linear-time greedy algorithm. Can we derive this algorithm?

5.1 Specification

Figure 11 summarises the specification, using some utility definitions in Figure 12. Time can be represented by some totally ordered numerical datatype with decidable comparison, and here we simply choose $\mathbb{N}$. An activity shall not finish before it

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

$$
\begin{array}{lcl}
\mathit{outr} & : & \{A\ B : \mathit{Set}\} \to (B \leftarrow (A \times B)) \\
\mathit{outr} & = & \mathit{fun}\ \mathit{proj}_2
\end{array}
$$

$$
\begin{array}{lcl}
\_¿\_ & : & \{A : \mathit{Set}\} \to \mathbb{P}\,A \to (A \leftarrow A) \\
(p\,¿)\ a'\ a & = & (a \equiv a') \times p\ a
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{all} & : & \{A : \mathit{Set}\} \to \mathbb{P}\,A \to \mathbb{P}\,(\mathsf{List}\ A) \\
\mathit{all}\ p\ [\,] & = & \top \\
\mathit{all}\ p\ (a :: \mathit{as}) & = & p\ a \times \mathit{all}\ p\ \mathit{as}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{check} & : & \{A : \mathit{Set}\} \to ((A \times \mathsf{List}\ A) \leftarrow (A \times \mathsf{List}\ A)) \to (\mathsf{List}\ A \leftarrow \mathsf{List}\ A) \\
\mathit{check}\ C & = & \mathit{foldR}\ (\mathit{cons} \circ C)\ \mathit{nil}
\end{array}
$$

Fig. 12. Some utility definitions.

starts; therefore, *Act* is a dependent pair consisting of a pair of start and finish times and a proof that the former is smaller than the latter. We also define auxiliary functions $\mathit{start} = \mathit{proj}_1 \cdot \mathit{proj}_1$ and $\mathit{finish} = \mathit{proj}_2 \cdot \mathit{proj}_1$. Two activities are disjoint from each other if one finishes before the other starts, as defined by the relation *disjoint*. The predicate (a set) *compatible* on (a, xs) checks whether a is compatible with all activities in xs. It is defined using *all*, which checks whether all elements in a list satisfy a given predicate.

A relation is *coreflexive* if it is a sub-relation of id_R. It is often used to filter those inputs that have properties we want. The operator $\_¿$ converts a predicate to a coreflexive relation. The relation $\mathit{check}\ C$ applies C to every head–tail decomposition within the input list. Note that if C is coreflexive, then so is $\mathit{check}\ C$.

The coreflexive relation *mutex*, defined using *check*, allows a list of activities to go through only if all members in the list are disjoint with each other. Also defined using *check* is *fin-ordered*, allowing through only those lists of activities that are ordered by finish time. Denoting activities by pairs, the list $(5, 7) :: (4, 5) :: (1, 3) :: [\,]$, for example, is a valid input. (It could have looked more natural had we used a snoc list, but we decided to stick with the existing datatype.) Its definition uses the predicate *fin-ubound*.

The relation *subseq*, defined as a fold, maps a list to one of its subsequences. Here the union of two relations is defined by $(R \sqcup S)\ a\ b = R\ a\ b \uplus S\ a\ b$.

Having these auxiliary relations defined, the specification *act-sel-spec* can be given in one line. The domain is restricted to finish-time sorted lists of activities by *fin-ordered*. We collect the set of all mutually compatible sub-lists of activities by $\Lambda(\mathit{mutex} \circ \mathit{subseq})$ and, using the relation $\leqslant_l$ that compares lists by their lengths, pick a longest solution.

5.2 The greedy theorem

Assume that we have molded an optimisation problem into the form $\mathit{max} \trianglelefteq \circ \Lambda(\mathit{foldR}\ S\ s)$. That is we want to pick a maximum under the ordering $\trianglelefteq$ from a set of solutions generated by a fold. The key step of the derivation is to transform the specification, whose direct interpretation implies generating very many solutions

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

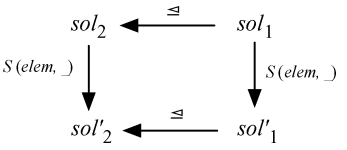

Fig. 13. The monotonicity condition.

before finally picking the best one, into an algorithm that greedily picks the local optimum at each step of the fold.

Consider two solutions sol_1 and sol_2 such that $sol_2 \trianglelefteq sol_1$. A relation $S : B \leftarrow (A \times B)$ is said to be *monotonic on* $\trianglelefteq : B \leftarrow B$ if, for all sol'_2 that is a possible result of $S\ (elem, sol_2)$, there exists some sol'_1 that is a result of $S\ (elem, sol_1)$ for which $sol'_2 \trianglelefteq sol'_1$ still holds, as illustrated in Figure 13.

If the monotonicity condition holds, nothing useful is going to emerge from the smaller solution sol_2, since for any sol'_2 there always exists a larger solution sol'_1; for any sol''_2 there always exists a larger solution sol''_1; and so on. Intuitively, at every step of the fold we only need to keep the best local solution.

Indeed, this is proved by Bird & de Moor (1997) as the greedy theorem. A relation S is monotonic on $\trianglelefteq$ if $S \circ (id_R \mathbin{\curlyvee} \trianglelefteq) \sqsubseteq \trianglelefteq \circ S$. Pointwise, the definition expands to (after some simplification)

$$\begin{array}{l}\forall\ sol'_2\ elem\ sol_1 \to \exists\ (\lambda sol_2 \to S\ \ sol'_2\ (elem, sol_2) \times sol_2 \trianglelefteq sol_1) \to \\ \quad \exists\ (\lambda sol'_1 \to S\ \ sol'_1\ (elem, sol_1) \times sol'_2 \trianglelefteq sol'_1).\end{array}$$

Since for all p and q we have $((\exists(\lambda x \to p\ x)) \to q) \Leftrightarrow \forall\ x \to p\ x \to q$, the above is equivalent to

$$\begin{array}{l}\forall\ sol'_2\ elem\ sol_1\ sol_2 \to (S\ \ sol'_2\ (elem, sol_2) \times sol_2 \trianglelefteq sol_1) \to \\ \quad \exists\ (\lambda sol'_1 \to S\ \ sol'_1\ (elem, sol_1) \times sol'_2 \trianglelefteq sol'_1),\end{array}$$

which meets our intuition in Figure 13. The greedy theorem is then given by [4]

$$\begin{array}{l}greedy\text{-}thm : \{A\ B : Set\}\{S : B \leftarrow (A \times B)\}\{s : \mathbb{P}\ B\}\{R : B \leftarrow B\} \to \\ \quad R \circ R \sqsubseteq R \ \ \to \ \ S \circ (id_R \mathbin{\curlyvee} R) \sqsubseteq R \circ S \ \ \to \\ \qquad foldR\ (max\ \ R\ {}_1\circ\ \Lambda S)\ (\Lambda(max\ \ R)\ s) \sqsubseteq max\ \ R\ {}_1\circ\ \Lambda(foldR\ \ S\ \ s).\end{array}$$

Note that the argument $R \circ R \sqsubseteq R$ requires that R be transitive. Both relations in the last line have type $B \leftarrow \mathsf{List}\ A$. The right-hand side of $\_\sqsubseteq\_$ is the specification, while the left-hand side picks the local optimal solution at each step.

5.3 Derivation using a partial greedy theorem

We attempt to solve the activity-selection problem by using the greedy theorem. Correspondingly, step 1 of the main derivation given in Figure 14 attempts to put the problem into the form $(max\ \trianglelefteq\ {}_1\circ\ \Lambda(foldR\ \ S\ \ e)) \circ \cdots$ for some $\trianglelefteq$, S and e. The

[4] For the purpose of this paper we formulate the theorem in terms of *max* rather than *min* as in Bird & de Moor (1997).

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

$$
\begin{array}{ll}
 & \mathit{act\text{-}sel\text{-}der} \;:\; \exists(\lambda f \to \mathit{fun}\, f \circ \mathit{fin\text{-}ordered} \sqsubseteq \mathit{act\text{-}sel\text{-}spec}) \\
 & \mathit{act\text{-}sel\text{-}der} \;=\; (\_,\, (\\
 & \quad (\mathit{max} \leq_{l\,1} \circ\, \Lambda(\mathit{mutex} \circ \mathit{subseq}))\ \circ \mathit{fin\text{-}ordered} \\
1: & \quad \sqsupseteq\langle\ \circ\text{-}\mathit{mono\text{-}l}\ (\mathit{min}\Lambda\text{-}\mathit{cong}\text{-}\sqsupseteq\ \mathit{mutex}\circ\mathit{subseq\text{-}is\text{-}fold})\ \rangle \\
 & \quad (\mathit{max} \leq_{l\,1} \circ\, \Lambda(\mathit{foldR}\ (\mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\,¿))\ \mathit{nil}))\ \circ \mathit{fin\text{-}ordered} \\
2: & \quad \sqsupseteq\langle\ \circ\text{-}\mathit{mono\text{-}l}\ (\mathit{max\text{-}mono}\ \trianglelefteq\text{-}\mathit{refines}\text{-}\leq_l)\ \rangle \\
 & \quad (\mathit{max} \trianglelefteq_1 \circ\, \Lambda(\mathit{foldR}\ (\mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\,¿))\ \mathit{nil}))\ \circ \mathit{fin\text{-}ordered} \\
3: & \quad \sqsupseteq\langle\ \mathit{fin\text{-}ubound\text{-}promotion}\ \rangle \\
 & \quad (\mathit{max} \trianglelefteq_1 \circ\, \Lambda(\mathit{foldR}((\mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\,¿)) \circ \mathit{fin\text{-}ubound}\,¿)\ \mathit{nil}))\ \circ \mathit{fin\text{-}ordered} \\
4: & \quad \sqsupseteq\langle\ \mathit{partial\text{-}greedy\text{-}thm}\ \ \trianglelefteq\text{-}\mathit{trans}\ \ \mathit{fin\text{-}ubound}¿ \sqsubseteq \mathit{id}_R\ \ \mathit{fin\text{-}ubound}¿ \sqsubseteq \mathit{id}_R \\
 & \qquad\qquad \mathit{monotonicity}\ \ \mathit{fin\text{-}ubound\text{-}homo}\ \rangle \\
 & \quad \mathit{foldR}\ (\mathit{max} \trianglelefteq_1 \circ\, \Lambda((\mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\,¿))\ \circ \mathit{fin\text{-}ubound}\,¿))\ \ (\Lambda(\mathit{max} \trianglelefteq)\ \mathit{nil}) \\
 & \qquad \circ \mathit{fin\text{-}ordered} \\
5: & \quad \sqsupseteq\langle\ \circ\text{-}\mathit{mono\text{-}l}\ (\mathit{foldR\text{-}mono}\ \ \sqsubseteq\text{-}\mathit{refl}\ \ \mathit{max}\text{-}\trianglelefteq\text{-}\mathit{nil} \sqsupseteq \mathit{nil})\ \rangle \\
 & \quad \mathit{foldR}\ (\mathit{max} \trianglelefteq_1 \circ\, \Lambda((\mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\,¿))\ \circ \mathit{fin\text{-}ubound}\,¿))\ \mathit{nil}\ \circ \mathit{fin\text{-}ordered} \\
6: & \quad \sqsupseteq\langle\ \circ\text{-}\mathit{mono\text{-}l}\ \mathit{algebra\text{-}refinement}\ \rangle \\
 & \quad \mathit{foldR}\ (\mathit{fun}\ (\mathit{uncurry}\ \mathit{compat\text{-}cons}) \circ \mathit{fin\text{-}ubound}\,¿)\ \mathit{nil}\ \circ \mathit{fin\text{-}ordered} \\
7: & \quad \sqsupseteq\langle\ \mathit{fin\text{-}ubound\text{-}demotion}\ \rangle \\
 & \quad \mathit{foldR}\ (\mathit{fun}\ (\mathit{uncurry}\ \mathit{compat\text{-}cons}))\ \mathit{nil}\ \circ \mathit{fin\text{-}ordered} \\
8: & \quad \sqsupseteq\langle\ \circ\text{-}\mathit{mono\text{-}l}\ (\mathit{foldR\text{-}to\text{-}foldr}\ \mathit{compat\text{-}cons}\ [])\ \rangle \\
 & \quad \mathit{fun}\ (\mathit{foldr}\ \mathit{compat\text{-}cons}\ [])\ \circ \mathit{fin\text{-}ordered} \\
 & \quad \sqsupseteq \Box))
\end{array}
$$

Fig. 14. The main derivation for the activity-selection problem.

lemma $\mathit{mutex} \circ \mathit{subseq}\text{-}\mathit{is}\text{-}\mathit{fold}$ helps turning $\mathit{mutex} \circ \mathit{subseq}$ into a fold by fold fusion. The fold uses a step relation $S = \mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\,¿)$, adding a new activity to a list only if they are compatible.

The relation S is not monotonic on $\leqslant_l$ – it is not always good to pick as many activities as possible, since some of them may be incompatible with activities to appear later. A typical strategy is to pick a relation stronger than $\leqslant_l$. Let activity a be *post-compatible* to xs if a not only is compatible with all activities in xs but also finishes later than them:

$$
\begin{array}{l}
\mathit{post\text{-}compatible} : \mathit{Act} \leftarrow \mathsf{List}\ \mathit{Act} \\
\mathit{post\text{-}compatible}\ a\ xs \;=\; \mathit{fin\text{-}ubound}\ (a, xs) \times \mathit{compatible}\ (a, xs).
\end{array}
$$

We define the ordering $\trianglelefteq$:

$$
\begin{array}{l}
xs \trianglelefteq ys \;=\; xs <_l ys \;\uplus \\
\qquad (xs \equiv_l ys \;\times\; \forall\, a \to \mathit{post\text{-}compatible}\ a\ xs \to \mathit{post\text{-}compatible}\ a\ ys),
\end{array}
$$

where $xs <_l ys$ if the length of xs is strictly smaller than that of ys, and $xs \equiv_l ys$ if they are equally long. Intuitively speaking, a list of activities ys is considered as good as xs under $\trianglelefteq$ if one of the two conditions holds. Possibly, ys contains strictly more activities than xs, since even if the next activity is in conflict with ys, it still has at least as many activities as $a :: xs$. Alternatively, ys and xs have the same size, but any activity that can be added to xs can also be added to ys.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

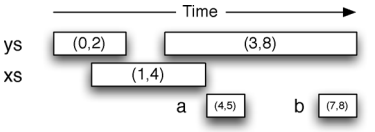

Fig. 15. A counterexample to the unconstrained monotonicity condition.

The definition of $\trianglelefteq$ can also be written in point-free style as

$$\begin{array}{lcl} \trianglelefteq & : & \mathsf{List}\ Act \leftarrow \mathsf{List}\ Act \\ \trianglelefteq & = & <_l \sqcup (\equiv_l \sqcap (post\text{-}compatible \backslash post\text{-}compatible)). \end{array}$$

Clearly, $\trianglelefteq$ is a subset of $\leqslant_l$, proved by the lemma $\trianglelefteq\text{-}refines\text{-}\leqslant_l \ : \trianglelefteq \sqsubseteq \leqslant_l$. In the second step of the derivation, lemma *max-mono* allows us to refine $max\ R$ to $max\ S$ if $S \sqsubseteq R$.

To apply the greedy theorem, we have to prove the monotonicity condition $S \circ (id_R \mathbin{\curlyvee} \trianglelefteq) \sqsubseteq \trianglelefteq \circ S$, which can be expanded to

$$\left.\begin{array}{l} \forall\ ys\ zs\ a\ xs \rightarrow \\ \quad xs \trianglelefteq ys\ \times\ (zs \equiv xs\ \uplus\ (zs \equiv a :: xs\ \times\ compatible\ (a, xs)) \rightarrow \\ \qquad zs \trianglelefteq ys\ \uplus\ (zs \trianglelefteq a :: ys\ \times\ compatible\ (a, ys)). \end{array}\right\} \quad (1)$$

However, this is *not* true in general. For a counterexample, consider $ys = (3, 8) :: (0, 2) :: []$, $xs = (1, 4) :: []$ and $a = (4, 5)$, as shown in Figure 15. We have $xs \trianglelefteq ys$ because ys contains more activities. Pick $zs = a :: xs$. In the last line of (1), a is not compatible with ys. However, $a :: xs \trianglelefteq ys$ is not true either, since $b = (7, 8)$, for example, can be safely added to $a :: xs$ but not to ys. It thus appears that we cannot safely drop xs.

In the context of our specification, this is a situation that should not have happened: we have required that the input list be sorted by finish time; thus a cannot finish earlier than the latest activity in ys. Such information is lost in the monotonicity condition. Note that it does not help to use a partial *cons* that builds sorted lists only. It would put restrictions between a and xs (indeed, $a :: xs$ in Figure 15 is sorted), but we need constraints on a and ys.

Many papers on program derivation take an informal approach when it comes to partiality of input. One would be allowed, rightfully, to informally convince the readers in words that (1) 'is true because the input is sorted,' because the reader need not know every detail. When it is Agda that we are trying to convince, however, one cannot get away with sloppiness. Agda keeps us honest.

To remedy the situation, it appears to be the right time to switch to indexed lists and make our derived program dependently typed as well. One could, for example, index the lists of activities with their finishing times. In our initial experiments, however, we found it harder than expected to mix indexed datatypes with this point-free style of program construction and would defer it to a possible future work.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

Instead, we proved a variation of the greedy theorem taking partiality into account:

$$
\begin{aligned}
&\mathit{partial\text{-}greedy\text{-}thm} : \{A\ B : \mathit{Set}\}\{S : B \leftarrow (A \times B)\}\{s : \mathbb{P}\, B\}\{R : B \leftarrow B\} \rightarrow \\
&\{C : (A \times \mathsf{List}\ A) \leftarrow (A \times \mathsf{List}\ A)\}\{D : (A \times B) \leftarrow (A \times B)\} \rightarrow \\
&\quad R \circ R \sqsubseteq R \ \rightarrow\ C \sqsubseteq \mathit{id}_R \ \rightarrow\ D \sqsubseteq \mathit{id}_R \ \rightarrow \\
&\quad S \circ (\mathit{id}_R \between R) \circ D \sqsubseteq R \circ S \ \rightarrow\ (\mathit{id}_R \between \mathit{foldR}\ S\ s) \circ C \sqsubseteq D \circ (\mathit{id}_R \between \mathit{foldR}\ S\ s) \rightarrow \\
&\qquad \mathit{foldR}\ (\mathit{max}\ R\ {}_1\circ \Lambda S)\ (\Lambda(\mathit{max}\ R)\ s)\ \circ\ \mathit{check}\ C \\
&\qquad\quad \sqsubseteq (\mathit{max}\ R\ {}_1\circ \Lambda(\mathit{foldR}\ S\ s))\ \circ\ \mathit{check}\ C.
\end{aligned}
$$

Comparing with the original theorem, the new ingredients include two coreflexive relations C and D (and preconditions $C \sqsubseteq \mathit{id}_R$ and $D \sqsubseteq \mathit{id}_R$, stating that they are indeed coreflexive). The argument $(\mathit{id}_R \between \mathit{foldR}\ S\ s) \circ C \sqsubseteq D \circ (\mathit{id}_R \between \mathit{foldR}\ S\ s)$ will be explained later.

The new monotonicity condition demanded by *partial-greedy-thm* is $S \circ (\mathit{id}_R \between R) \circ D \sqsubseteq R \circ S$. It is easier to see the roles of C and D if we instantiate the new monotonicity condition to the activity-selection problem,

$$
\begin{aligned}
&\mathit{monotonicity}\ : \\
&\quad ((\mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\ ¿)) \circ \mathit{fin\text{-}ubound}\ ¿) \circ (\mathit{id}_R \between \trianglelefteq) \circ \mathit{fin\text{-}ubound}\ ¿ \ \sqsubseteq \\
&\qquad \trianglelefteq \circ (\mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\ ¿)) \circ \mathit{fin\text{-}ubound}\ ¿,
\end{aligned}
$$

which expands to

$$
\begin{aligned}
&\forall\ \mathit{ys}\ \mathit{zs}\ a\ \mathit{xs} \rightarrow (\mathit{zs} \equiv \mathit{xs}\ \uplus\ (\mathit{zs} \equiv a :: \mathit{xs}\ \times\ \mathit{compatible}\ (a, \mathit{xs})) \times \\
&\quad \mathit{fin\text{-}ubound}\ (a, \mathit{xs})\ \times\ \mathit{xs} \trianglelefteq \mathit{ys}\ \times\ \mathit{fin\text{-}ubound}\ (a, \mathit{ys})\ \rightarrow \\
&\qquad (\mathit{zs} \trianglelefteq \mathit{ys}\ \uplus\ (\mathit{zs} \trianglelefteq a :: \mathit{ys}\ \times\ \mathit{compatible}\ (a, \mathit{ys}))) \times \mathit{fin\text{-}ubound}\ (a, \mathit{ys}).
\end{aligned}
$$

We pick the step relation S to be $(\mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\ ¿)) \circ \mathit{fin\text{-}ubound}\ ¿$. The coreflexive relation D is instantiated to $\mathit{fin\text{-}ubound}\ ¿$, checking that a does finish before all activities in ys. The coreflexive relation C, on the other hand, is a variation of D that can be checked before $\mathit{fold}\ S\ s$. The condition $(\mathit{id}_R \between \mathit{foldR}\ S\ s) \circ C \sqsubseteq D \circ (\mathit{id}_R \between \mathit{foldR}\ S\ s)$ basically says that checking C before the fold guarantees that D holds after the fold. For the activity-selection problem, we pick C to be $\mathit{fin\text{-}ubound}\ ¿$ as well.

Finally, the input is filtered by $\mathit{check}\ C$ (definition given in Figure 12), which checks that C holds for every tail of the input list. For the activity-selection problem, $\mathit{check}\ C$ equals *fin-ordered*.

Back to the derivation. In step 3, *fin-ubound-promotion* combines several lemmas that use properties of coreflexive relations to promote *fin-ubound* into the fold. We are then ready to use the partial greedy theorem in step 4, with the monotonicity condition given above.

Steps 5–7 refine the argument of *foldR* to a function, using the fact that $\mathit{max} \trianglelefteq {}_1\circ \Lambda(\mathit{outr} \sqcup (\mathit{cons} \circ \mathit{compatible}\ ¿))$ can be refined to the function *compat-cons*, defined by

$$
\begin{aligned}
&\mathit{compat\text{-}cons}\ :\ \mathit{Act} \rightarrow \mathsf{List}\ \mathit{Act} \rightarrow \mathsf{List}\ \mathit{Act} \\
&\mathit{compat\text{-}cons}\ a\ [\,]\ =\ a :: [\,] \\
&\mathit{compat\text{-}cons}\ a\ (x :: \mathit{xs})\ \mathbf{with}\ \mathit{finish}\ x \leqslant ?\ \mathit{start}\ a \\
&\ldots \mid \mathit{yes}\ \_\ =\ a :: x :: \mathit{xs} \\
&\ldots \mid \mathit{no}\ \_\ =\ x :: \mathit{xs},
\end{aligned}
$$

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

where $\_\leqslant?\_$ compares its two arguments and returns either *yes* or *no*, which is pattern matched by the **with** construct. While **with** (McBride & McKinna 2004) is an important construct that refines types in the context as well as the value being inspected, for this example, it functions like a **case** expression in a typical functional language.

Finally in step 8, the relational fold can be refined into a functional *foldr*, using the following lemma *foldR-to-foldr*:

$$\begin{aligned} \mathit{foldR\text{-}to\text{-}foldr} \;:\; \{A\ B : \mathit{Set}\} \to (f : A \to B \to B) \to (e : B) \to \\ \mathit{foldR}\ (\mathit{fun}\ (\mathit{uncurry}\ f))\ (\mathit{singleton}\ e) \sqsupseteq \mathit{fun}\ (\mathit{foldr}\ f\ e). \end{aligned}$$

We have thus derived an algorithm for the activity-selection problem: *foldr compat-cons* []. The actual derivation consists of about 550 lines of Agda code.

6 Unfolds and hylomorphism

While many algorithms can be expressed as folds, there are cases in which we need general recursion. It is folklore knowledge that general recursion can be expressed as a *hylomorphism* (Meijer *et al.* 1991): a fold after an unfold. The unfolding phase expands a data structure corresponding to the call tree of the general recursive function one wants to define, while the folding phase eliminates the tree and processes the results from the recursive calls. The intermediate data structure can be removed through a process called *deforestation* (Wadler 1990). It is such a fundamental transformation that people in the program derivation community sometimes do not distinguish the deforested program and the hylomorphic program with the intermediate tree. Hylomorphism is definable, provided that inductive types and coinductive types coincide. This also introduces potential non-termination, and a semantics based on sets and total functions no longer suffices. The usual solution is to move to a semantics, using complete partial orders and continuous functions.

However, many hylomorphic algorithms we intend to construct do terminate. For this type of program derivation, it is preferable to stay within a simple semantics, using sets and total functions. Indeed, Bird & de Moor (1997) used exclusively folds on inductive types. The 'unfolding' phase of a hylomorphism is modelled using relational converse of a fold. Successive seeds in the unfolding phase are related by a *well-founded* relation; that is there exists no infinitely descending chain; therefore the unfolding must eventually terminate. Termination also guarantees uniqueness of solution; therefore terminating general recursion gives valid definitions (Doornbos & Backhouse 1996). A theory relating well-foundedness, induction and termination has been thoroughly studied by Doornbos & Backhouse (1995, 1996). For the rest of the paper, when we say 'unfold' we refer to converse of a relational fold on an inductive type or its functional refinement, rather than unfold for coinductive types.

The next question is how to model the theory in a dependently typed programming language. Like many such languages, Agda distinguishes between inductive and

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

coinductive types. For inductively defined functions, Agda deploys a termination check based on structural recursion. Agda knows, for example, that *foldr* terminates because the argument xs passed to the recursive call is a substructure of $x :: xs$. Consider, however, the typical algorithm computing the greatest common divisor by mutually subtracting the two given numbers. We need some extra mechanisms to convince Agda, which recognises only structural recursion, that the subtracted number does get 'smaller'.

In the next few sections we will demonstrate how to encode user-defined notions of order and well-foundedness into the structure-based termination check of Agda. To derive an algorithm using unfold, the programmer proceeds with the derivation in the relational setting and provides a proof of termination in the last step when the converse of a fold is eventually refined to a functional unfold.

6.1 Well-founded recursion and inductivity

The notion of well-founded relation has deep root in mathematics. This section attempts to build a connection to theories in relational program construction. For a type theoretical view, the readers are referred to Nordström (1988) and the like. More references are given in Section 7.

Recall, as explained in Section 4.4, that $R \backslash S$ is the largest relation such that $R \circ R \backslash S$ is still contained in S. The *monotype factor* is a related notion defined on sets.[5] Given a relation $R : B \leftarrow A$ and a subset s of B, $R \mathbin{\backslash\!\cdot} s$ is the largest set such that when the domain of R is restricted to $R \mathbin{\backslash\!\cdot} s$, its range is still in s:

$$\begin{aligned} &\_ \mathbin{\backslash\!\cdot} \_ : \{A\ B : Set\} \to (B \leftarrow A) \to \mathbb{P}\, B \to \mathbb{P}\, A \\ &(R \mathbin{\backslash\!\cdot} s)\ a = \forall\ b \to R\ b\ a \to s\ b. \end{aligned}$$

A relation $R : A \leftarrow A$ is said to be *inductive*[6] if for all $s : \mathbb{P}\, A$,

$$R \mathbin{\backslash\!\cdot} s \subseteq s \quad \Rightarrow \quad A \subseteq s, \tag{2}$$

where A is the set of all elements having type A.

One of the ways to understand (2) is to expand the definitions of $\_ \mathbin{\backslash\!\cdot} \_$, $\_ \subseteq \_$ and $\_ \Rightarrow \_$:

$$(\forall\ a \to (\forall\ b \to R\ b\ a \to s\ b) \to s\ a) \to (\forall\ a \to s\ a).$$

It is the principle of strong induction: if given proofs of $s\ b$ for all b's that are 'smaller than' a with respect to R, one may prove $s\ a$, then s holds for all a's. An inductive relation, as the name suggests, is one that we may use to perform strong induction with. (It is different from an 'inductively defined relation'.) Doornbos & Backhouse (1995, 1996) showed that inductivity is the notion that captures program termination.

[5] To be more precise, $¡(R \mathbin{\backslash\!\cdot} s) = R \backslash (¡s)$, where $¡\_ : \mathbb{P}\, A \to (A \leftarrow \top)$ is the isomorphism between sets of A and relations from $\top$ to A.

[6] Bird & de Moor (1997) gave a definition of inductivity using $\_ \backslash \_$, which is shown by Doornbos (1996) to be equivalent to the definition here.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

How do we model inductivity in Agda? Another way to look at (2) is that it demands the set of all A's to be the least fixed point of the function $(\lambda X \rightarrow R \backslash X)$. To create a least fixed point, naturally, we use a **data** declaration:

$$\begin{array}{l}\textbf{data}\ \ \mathsf{Acc}\ \{A : Set\}(R : A \leftarrow A) : A \rightarrow Set\ \ \textbf{where}\\ \quad \mathsf{acc}\ \ :\ (R \backslash \mathsf{Acc}\ R)\ \subseteq\ \mathsf{Acc}\ R.\end{array}$$

Therefore, acc is a proof that $(R \backslash \mathsf{Acc}\ R)\ \subseteq\ \mathsf{Acc}\ R$. A relation R is inductive if for every $x : A$, we can prove $\mathsf{Acc}\ R\ x$.

However, if we expand the definition of Acc,

$$\begin{array}{l}\textbf{data}\ \mathsf{Acc}\ \{A : Set\}(\_\lhd\_ : A \leftarrow A) : A \rightarrow Set\ \textbf{where}\\ \quad \mathsf{acc}\ \ :\ \forall\ x \rightarrow (\forall\ y \rightarrow y \lhd x \rightarrow \mathsf{Acc}\ \_\lhd\_\ y) \rightarrow \mathsf{Acc}\ \_\lhd\_\ \ x,\end{array}$$

then this is exactly the definition of *accessibility* described by, for example, Nordström (1988). Consider a strict partial order $\_\lhd\_$ on a set A. For all minimal x in A, since there exists no y such that $y \lhd x$, the proposition $(\forall\ y \rightarrow y \lhd x \rightarrow \mathsf{Acc}\ \_\lhd\_\ y)$ is satisfied, and we can thus construct proofs for $\mathsf{Acc}\ \_\lhd\_\ x$. We can then construct $\mathsf{Acc}\ \_\lhd\_\ x_1$ for x_1 whose predecessors are all such minimal x's and so on. Having a proof of $\mathsf{Acc}\ \_\lhd\_\ \ x_n$ means that if we follow a descending chain $x_n \rhd x_{n-1} \rhd x_{n-2} \ldots$, it has to stop at some base case – a property usually referred to as *well-foundedness.*

Note that every incomparable element is trivially accessible. If the relation $\_\lhd\_$ is empty, all elements are accessible because they are all minimal.

The datatype Acc echoes the observation of Bird & de Moor (1997) that inductivity and well-foundedness are equivalent concepts in the category of sets and relations. As stated before, a relation R is inductive, or well founded, if every $x : A$ is in $\mathsf{Acc}\ R$:

$$\begin{array}{lcl}\mathit{well\text{-}found} & : & \{A : Set\} \rightarrow (A \leftarrow A) \rightarrow Set\\ \mathit{well\text{-}found}\ R & = & \forall\ \ x \rightarrow \mathsf{Acc}\ R\ x.\end{array}$$

Remark: Doornbos & Backhouse (1995, 1996) generalised inductivity to arbitrary F-functors and called it F-*reductivity*. In Agda, however, we find it easier to construct membership relations (see the next section) than to parameterise Acc with a functor. Also, they defined 'F-well-foundedness to be 'having a unique solution', which they proved to be strictly weaker than F-reductivity.

6.2 Example: 'Less-than is well founded

Recall the definitions of $\mathbb{N}$, $\_\leqslant\_$ and $\_<\_$ in Figure 2, where the base case $\leqslant$-refl states that $\_\leqslant\_$ is reflexive, while the recursive case $\leqslant$-step concludes $m \leqslant \mathsf{suc}\ n$ from a proof of $m \leqslant n$. The less-than relation is defined in terms of $\_\leqslant\_$:

$$\begin{array}{lcl}\_<\_ & : & \mathbb{N} \leftarrow \mathbb{N}\\ m < n & = & \mathsf{suc}\ m \leqslant n\end{array}$$

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

One can show that $\_<\_$ is well founded:

$$
\begin{array}{lll}
\mathbb{N}\text{-}wf & : & well\text{-}found\ \_<\_ \\
\mathbb{N}\text{-}wf\ n & = & \mathsf{acc}\ n\ (access\ n) \\
\quad \textbf{where}\ access & : & (n : \mathbb{N}) \to \forall\ m \to m < n \to \mathsf{Acc}\ \_<\_\ m \\
\qquad access\ \ \mathsf{zero} \quad m\ \ () & & \\
\qquad access\ \ .(\mathsf{suc}\ m)\ \ m\ \ \leqslant\text{-}\mathsf{refl} & = & \mathsf{acc}\ m\ (access\ m) \\
\qquad access\ \ (\mathsf{suc}\ n)\ \ m\ \ (\leqslant\text{-}\mathsf{step}\ m{<}n) & = & access\ n\ m\ m{<}n.
\end{array}
$$

The main work is done in the auxiliary function *access*, whose job, given a fixed n, is to construct an accessibility proof for all m that is smaller than n. In the case in which n is zero, there can be no proof of $m < n$. The pair of parentheses () in the first base case, the *absurd pattern*, is Agda's syntax stating that the type $m <$ zero is empty; therefore, this case could not happen in any well-typed program (Goguen *et al.* 2006).

The next two cases deal with non-zero n. For the case of $\leqslant$-step $m{<}n$, the proof is deconstructed and passed to the recursive call. We keep doing so until we reach the base case $\leqslant$-refl. For an operational explanation, the proof shows that we can always reach m from n in a finite number of steps by going through the proof of $m < n$. Hence the name 'accessibility'.

When we finally reach the base case $\leqslant$-refl, n is unified with suc m. The 'dot pattern' indicates that the only possible value of the first argument is suc m. For this case we shall return a proof of Acc $\_<\_$ m, which we do by returning a constructor acc. The second argument to acc shall be a function having type $\forall\ k \to k < m \to \mathsf{Acc}\ \_<\_\ k$, which we can construct by calling *access* m.

6.3 Hylomorphism in Agda

Now we are able to discuss how terminating unfolds and hylomorphisms can be defined for a given inductive type. Typically, the generating function in an unfold takes a seed of type B and returns an $\mathsf{F}B$-structure, where F is the base functor of the generated datatype. As an example, consider the datatype for internally labelled binary tree, defined in Figure 16. Unfold for Tree takes a generating function that given an element of B, returns a structure of type $\top \uplus (A \times B \times B)$. For first-ordered B, we can define a *membership relation* that extracts the B-typed new seeds in the returned structure. For Tree, we define:

$$
\begin{array}{lllll}
\varepsilon\text{-}TreeF & : & \{A\ B : Set\} \to (B \leftarrow (\top \uplus (A \times B \times B))) & & \\
\varepsilon\text{-}TreeF & (\mathsf{inj}_1\ \_) & \_ & = & \bot \\
\varepsilon\text{-}TreeF & (\mathsf{inj}_2\ (a, b_1, b_2)) & b & = & (b_1 \equiv b)\ \uplus\ (b_2 \equiv b).
\end{array}
$$

Both Bird & de Moor (1997) and Doornbos & Backhouse (1995, 1996) showed that unfolding a tree using a relation $R : (\top \uplus (A \times B \times B)) \leftarrow B$ terminates if $\varepsilon\text{-}TreeF \circ R$ is inductive, which means that one cannot repeatedly apply R to the new seeds forever. This can be verified by constructing the following variation of tree unfold which, apart from a generating function f and a seed b, takes an additional

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

```
data Tree (A : Set) : Set where
  Null : Tree A
  Fork : A → Tree A → Tree A → Tree A

foldt : {A B : Set} → ((A × B × B) → B) → B → Tree A → B
foldt f e Null = e
foldt f e (Fork a t u) = f (a, foldt f e t, foldt f e u)

foldT : {A B : Set} → (B ← (A × B × B)) → ℙ B → (B ← Tree A)
foldT R s = ∈₁ ∘ foldt (Λ(R ∘ (id_R ╳ ∈ ╳ ∈))) s

foldT-fusion-⊒ : {A B C : Set}(R : C ← B) →
  {S : B ← (A × B × B)}{T : C ← (A × C × C)}{u : ℙ B}{v : ℙ C} →
    (R ∘ S) ⊒ (T ∘ (id_R ╳ R ╳ R)) → Λ(R ∘ ∈) u ⊇ v →
      (R ∘ foldT S u) ⊒ foldT T v
```

Fig. 16. Internally labelled binary tree and its fold. For clarity we pretend that *foldt* is universe polymorphic.

argument stating that b is accessible under the relation $\varepsilon\text{-}TreeF \circ fun\ f$:

```
unfoldt-acc : {A B : Set} → (f : B → ⊤ ⊎ (A × B × B)) →
                (b : B) → Acc (ε-TreeF ∘ fun f) b → Tree A
unfoldt-acc f b (acc .b h) with f b
... | inj₁ _ = Null
... | inj₂ (a, b₁, b₂) =
    Fork a (unfoldt-acc f b₁ (h b₁ (inj₂ (a, b₁, b₂), inj₁ ≡-refl, ≡-refl)))
           (unfoldt-acc f b₂ (h b₂ (inj₂ (a, b₁, b₂), inj₂ ≡-refl, ≡-refl))).
```

This is a special case of Bove & Capretta's (2005) approach of encoding general recursion in type theory. Apart from the third argument, *unfoldt-acc* is just like an ordinary unfold in a functional language that allows general recursion. In the two recursive calls, accessibility of b_1 and b_2 is obtained by applying h. The argument $(\mathsf{inj}_2\ (a, b_1, b_2), \mathsf{inj}_1\ \equiv\text{-refl}, \equiv\text{-refl})$ is simply a proof of $(\varepsilon\text{-}TreeF \circ fun\ f)\ b_1\ b$: that b_1 is one of the new seeds generated by $f\ b = \mathsf{inj}_2\ (a, b_1, b_2)$. This definition passes the termination check because application of h is considered structurally smaller than $\mathsf{acc}\ b\ h$. The accessibility proof is a structure of a finite depth; therefore the unfolding must terminate.

Now we can define *unfoldt*:

```
unfoldt : {A B : Set} → (f : B → ⊤ ⊎ (A × B × B)) →
                        well-found (ε-TreeF ∘ fun f) → B → Tree A
unfoldt f wf b = unfoldt-acc f b (wf b).
```

To generate a tree using *unfoldt*, one supplies a generating function f, a seed s and a proof wf of well-foundedness: that all values in the domain of f are accessible.

6.4 New accessibility from old

It is rather tedious, however, having to give a proof of accessibility from scratch every time. Doornbos & Backhouse (1996) suggested a methodology to construct

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

new accessibility arguments from old ones. The following two lemmas will be particularly useful for us. The first lemma states that if x is accessible under S, it is also accessible under its sub-relation:

$$\begin{array}{l} \mathit{acc}\text{-}\sqsubseteq : \{A : \mathit{Set}\}\{R\ S : A \leftarrow A\} \rightarrow R \sqsubseteq S \ \rightarrow\ \mathsf{Acc}\ S \subseteq \mathsf{Acc}\ R \\ \mathit{acc}\text{-}\sqsubseteq\ \{A\}\{R\}\ \ R \sqsubseteq S\ \ x\ \ (\mathsf{acc}\ \_ x\ h)\ =\ \mathsf{acc}\ x\ \mathit{access} \\ \quad \textbf{where}\ \ \mathit{access} : (y : A) \rightarrow R\ y\ x \rightarrow \mathsf{Acc}\ R\ y \\ \qquad\quad \mathit{access}\ y\ \mathit{yRx}\ =\ \mathit{acc}\text{-}\sqsubseteq\ \ R \sqsubseteq S\ y\ (h\ y\ (R \sqsubseteq S\ y\ x\ \mathit{yRx})). \end{array}$$

For example, given the following function *pred*, we may use *unfoldt* to generate a full binary tree counting down from the given seed, provided that we can prove the well-foundedness of ε-*TreeF* $\circ$ *fun pred*:

$$\begin{array}{lcl} \mathit{pred} & : & \mathbb{N} \rightarrow \top \uplus (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \\ \mathit{pred}\ \mathsf{zero} & = & \mathsf{inj}_1\ \mathit{tt} \\ \mathit{pred}\ (\mathsf{suc}\ n) & = & \mathsf{inj}_2\ (n, n, n). \end{array}$$

Rather than having it proven from scratch, recall that we have shown in the previous section that $\_<\_$ is well founded. If we can prove that

$$\mathit{pred} \sqsubseteq < \ :\ (\varepsilon\text{-}\mathit{TreeF}\ \circ\ \mathit{fun}\ \mathit{pred})\ \ \sqsubseteq\ \ \_<\_,$$

which simplifies to $\mathit{pred}\ n \equiv \mathsf{inj}_2(m, m_1, m_2) \times (k \equiv m_1 \uplus k \equiv m_2) \rightarrow k < n$, that is both seeds n_1 and n_2 returned by *pred* are smaller than the input n, we can conclude that ε-*TreeF* $\circ$ *fun pred* is well founded. The proof is trivial. We may then call *unfoldt* as follows:

$$\begin{array}{lcl} \mathit{down} & : & \mathbb{N} \rightarrow \mathsf{Tree}\ \mathbb{N} \\ \mathit{down} & = & \mathit{unfoldt}\ \mathit{pred}\ (\lambda x \rightarrow \mathit{acc}\text{-}\sqsubseteq\ \ \mathit{pred} \sqsubseteq <\ \ x\ \ (\mathbb{N}\text{<-}\mathit{wf}\ x)). \end{array}$$

Assume that R is the relation mapping the current seed x to the next seed x'. The next lemma corresponds to the typical proof of termination, using a bound function f and proving that $f\ x'$ is strictly smaller than $f\ x$ with respect to some ordering that is known to be well founded. That is the lemma establishes the well-foundedness of R, given the well-foundedness of $\mathit{fun}\ f \circ R \circ (\mathit{fun}\ f)^{\smile}$:

$$\begin{array}{l} \mathit{acc}\text{-}\mathit{fRf}^{\smile}\ :\ \{A\ B : \mathit{Set}\}\{R : A \leftarrow A\}\{f : A \rightarrow B\} \rightarrow \\ \qquad (x : A) \rightarrow \mathsf{Acc}\ \ (\mathit{fun}\ f \circ R \circ (\mathit{fun}\ f)^{\smile})\ (f\ x) \rightarrow \mathsf{Acc}\ R\ x \\ \mathit{acc}\text{-}\mathit{fRf}^{\smile}\ \ \{A\}\{B\}\{R\}\{f\}\ x\ (\mathsf{acc}\ \_\ h)\ =\ \mathsf{acc}\ x\ \mathit{access} \\ \quad \textbf{where}\ \ \mathit{access}\ :\ (y : A) \rightarrow R\ y\ x \rightarrow \mathsf{Acc}\ R\ y \\ \qquad\quad \mathit{access}\ y\ \mathit{yRx}\ =\ \mathit{acc}\text{-}\mathit{fRf}^{\smile}\ y\ (h\ (f\ y)\ (y, \equiv\text{-refl}, (x, \mathit{yRx}, \equiv\text{-refl}))). \end{array}$$

We will see an example in the next section.

6.5 Example: Deriving quicksort

For an example deriving a hylomorphism, let us see a derivation of quicksort, adopted from Bird (1996). The hylomorphism unfolds a binary search tree by pivoting from the input list, before flattening it to produce the sorted list. Deforesting the hylomorphism yields the familiar algorithm for quicksort. It is worth noting that

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

sorting via a binary search tree is the original example of structural recursion in Burstall (1969).

6.5.1 Specifying sort

We first specify what a sorted list is, assuming a datatype Val and a binary ordering $\_\leqslant\_ : \mathsf{Val} \to \mathsf{Val} \to \mathit{Set}$ that form a decidable total order. To begin with, let *lbound* be the set of all pairs (a, xs) such that a is a lower bound of xs:

$$\begin{aligned} &\mathit{lbound} \;:\; \mathbb{P}\,(\mathsf{Val} \times \mathsf{List\ Val}) \\ &\mathit{lbound}\ (a, xs) \;=\; \mathit{all}\ (\lambda b \to a \leqslant b)\ xs \end{aligned}$$

The coreflexive relation *ordered*?, which lets a list go through if and only if it is sorted, can then be defined by

$$\begin{aligned} &\mathit{ordered}? \;:\; \mathsf{List\ Val} \leftarrow \mathsf{List\ Val} \\ &\mathit{ordered}? \;=\; \mathit{check}\ (\mathit{lbound}\ ¿), \end{aligned}$$

where *check* is defined in Figure 12.

We postulate a type Bag, representing bags of values that are formed by two postulated functions $\wr\int : \mathsf{Bag}$ and $\_::_b\_ : \mathsf{Val} \to \mathsf{Bag} \to \mathsf{Bag}$. We demand that the result of $\_::_b\_$ be distinguishable from the empty bag and that $\_::_b\_$ be commutative:[7]

$$\begin{aligned} ::_b\text{-}\mathit{nonempty} \;&:\; \forall\ \{a\ w\} \to (\wr\int \equiv a ::_b w) \to \bot \\ ::_b\text{-}\mathit{commute} \;&:\; (a\ b : \mathsf{Val}) \to (w : \mathsf{Bag}) \to a ::_b (b ::_b w) \equiv b ::_b (a ::_b w). \end{aligned}$$

The function *bagify*, defined below, converts a list to a bag by a fold:

$$\begin{aligned} &\mathit{bagify} \;:\; \mathsf{List\ Val} \to \mathsf{Bag} \\ &\mathit{bagify} \;=\; \mathit{foldr}\ \_::_b\_\ \wr\int. \end{aligned}$$

To map a list to one of its arbitrary permutations, we simply convert it to a bag and convert the bag back to a list. To sort a list is to find one of its permutations that is sorted:

$$\begin{aligned} &\mathit{permute} \;:\; \mathsf{List\ Val} \leftarrow \mathsf{List\ Val} \\ &\mathit{permute} \;=\; (\mathit{fun}\ \mathit{bagify})^{\smile} \circ \mathit{fun}\ \mathit{bagify}, \end{aligned}$$

$$\begin{aligned} &\mathit{sort} \;:\; \mathsf{List\ Val} \leftarrow \mathsf{List\ Val} \\ &\mathit{sort} \;=\; \mathit{ordered}? \circ \mathit{permute}. \end{aligned}$$

Thus the specification is complete, from which we shall derive an algorithm that actually sorts a list.

6.5.2 The derivation

The main derivation is shown in Figure 17, while some of the lemmas needed are summarised in Figure 18. The first step we do is to introduce an id_R between

[7] The presentation here is simplified. In fact we postulated another equality for bags as well as its congruence and substitution rules. The details, however, are not relevant here.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

```
qsort-der : ∃ (λf → ordered? ∘ permute ⊒ fun f)
qsort-der = (_, (
    ordered? ∘ permute
1 :  ⊒⟨  ∘-mono-r  id-intro-l  ⟩
    ordered? ∘ id_R ∘ permute
2 :  ⊒⟨  ↤-mono (ordered?..) (fun flatten • (fun flatten)˘ ..) (id_R ..) fun-simple  ⟩
    ordered? ∘ fun flatten ∘ (fun flatten)˘ ∘ permute
3 :  ⊒⟨  ↤-mono-l (fun flatten • ordtree? ..) (ordered? • fun flatten ..) ordflatten  ⟩
    fun flatten ∘ ordtree? ∘ (fun flatten)˘ ∘ permute
4 :  ⊒⟨  ∘-mono-r  refine-converses  ⟩
    fun flatten ∘ ((permute ∘ fun flatten) ∘ ordtree?)˘
5 :  ⊒⟨  ∘-mono-r  (˘-monotonic-⇐ (foldT-fusion-⊒ (permute ∘ fun flatten) fuse1 fuse2))  ⟩
    fun flatten ∘ (foldT (permute ∘ fun join ∘ okl¿) nil)˘
6 :  ⊒⟨  ∘-mono-r  (˘-monotonic-⇐ (foldT-monotonic part1 part2))  ⟩
    fun flatten ∘ (foldT ((fun partition)˘ ∘ fun inj_2) (λb → isInj_1 (partition b)))˘
7 :  ⊒⟨  ∘-mono-r  (foldT-to-unfoldt partition partition-wf)  ⟩
    fun flatten ∘ fun (unfoldt partition partition-wf)
8 :  ⊒⟨  fun∘-⊒  ⟩
    fun (flatten · unfoldt partition partition-wf)
  ⊒□))
```

Fig. 17. The main derivation for quicksort.

```
fun-simple  :  {A B : Set}{f : A → B} → fun f ∘ (fun f)˘ ⊑ id_R
ordflatten  :  fun flatten ∘ ordtree? ⊑ ordered? ∘ fun flatten
fuse1       :  (permute ∘ fun join ∘ okl¿) ∘
                 (id_R ╳ (permute ∘ fun flatten) ╳ (permute ∘ fun flatten))
                   ⊑ (permute ∘ fun flatten) ∘ (fork ∘ okt¿)
fuse2       :  nil ⊆ Λ((permute ∘ fun flatten) ∘ ∈) null
fun∘-⊒      :  {A B C : Set}{g : B → C}{f : A → B} → fun g ∘ fun f ⊒ fun (g · f)
```

Fig. 18. Some lemmas used in the derivation of quicksort.

ordered? and *permute*, which is then split into *fun flatten* ∘ (*fun flatten*)˘ in step 2, using *fun-simple*. The lemma *fun-simple* is one of the properties that characterises a function: a function maps an input to at most one output; therefore *fun f* ∘ (*fun f*)˘ must map a value to itself. The function *flatten* is defined as a fold:

$$\begin{array}{lcl} \mathit{flatten} & : & \{A : \mathit{Set}\} \to \mathsf{Tree}\ A \to \mathsf{List}\ A \\ \mathit{flatten} & = & \mathit{foldt}\ \mathit{join}\ [] , \end{array}$$

where *join* $(a, xs, ys) = xs +\!\!+ (a :: ys)$. Combinators $\leftarrow\!\!\!\!\cdot$*-mono*, _.._ and _•_, are packaged applications of monotonicity and associativity. We need not go into their details here.

In step 3, *ordflatten* is used to transform the test of sortedness on lists to a test on trees. The coreflexive relation *ordtree*? is defined as a fold:

$$\begin{array}{lcl} \mathit{ordtree?} & : & \mathsf{Tree\ Val} \leftarrow \mathsf{Tree\ Val} \\ \mathit{ordtree?} & = & \mathit{foldT}\ (\mathit{fork} \circ \mathit{okt}\ ¿)\ \mathit{null}, \end{array}$$

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

where $fork = fun\ (\lambda(a, t, u) \to \mathsf{Fork}\ a\ t\ u)$; $null = singleton\ \mathsf{Null}$; and okt is a predicate on $(\mathsf{Val} \times \mathsf{Tree}\ \mathsf{Val} \times \mathsf{Tree}\ \mathsf{Val})$ defined by

$$\begin{aligned} okt\ (a, t, u) \quad = \quad & (\forall\ a' \to \varepsilon\text{-}Tree\ a'\ t \to a' \leqslant a) \times \\ & (\forall\ a' \to \varepsilon\text{-}Tree\ a'\ u \to a \leqslant a'), \end{aligned}$$

where $\varepsilon\text{-}Tree : A \leftarrow \mathsf{Tree}\ A$, relating a tree to one of its elements, can be defined using $foldT$.

The sub-proof *refine-converses* in step 4 uses properties of converses and coreflexive relations to group $(permute \circ fun\ flatten) \circ ordtree?$ together. This relation restricts its domain to trees that are ordered, flattens the tree and maps the resulting list to one of its permutations. Since $ordtree?$ is a fold, we apply $foldT\text{-}fusion\text{-}\sqsupseteq$ in step 5 to fuse them into a single $foldT$:

$$foldT\ (permute \circ fun\ join \circ okl\ ¿)\ nil. \tag{3}$$

The fusion conditions $fuse1$ and $fuse2$ are given in Figure 18. The predicate okl is similar to okt but is defined on lists. Given a tuple (a, xs, ys), the relation $permute \circ fun\ join \circ okl\ ¿$ checks whether a is no less than all elements in xs and no larger than all elements in ys and returns a permutation of $xs +\!\!+ (a :: ys)$.

Define the function *partition* as follows:

$$\begin{aligned} & partition\ :\ \mathsf{List}\ \mathsf{Val} \to (\top \uplus (\mathsf{Val} \times \mathsf{List}\ \mathsf{Val} \times \mathsf{List}\ \mathsf{Val})) \\ & partition\ [] \ = \ \mathsf{inj}_1\ tt \\ & partition\ (x :: xs) \ = \ \mathsf{inj}_2\ (x, split\ x\ xs), \end{aligned}$$

where the function call $split\ x\ xs$ splits the list xs into those that are smaller than or equal to x and those that are larger than x,

$$\begin{aligned} & split\ :\ \mathsf{Val} \to \mathsf{List}\ \mathsf{Val} \to (\mathsf{List}\ \mathsf{Val} \times \mathsf{List}\ \mathsf{Val}) \\ & split\ x\ [] \ = \ ([], []) \\ & split\ x\ (y :: xs)\ \mathbf{with}\ split\ x\ xs \\ & \ldots \ \mid (ys, zs)\ \mathbf{with}\ y \leqslant? x \\ & \ldots \qquad \mid yes\ \_ = (y :: ys, zs) \\ & \ldots \qquad \mid no\ \_ = (ys, y :: zs). \end{aligned}$$

It is possible to show that both arguments to $foldT$ can be expressed in terms of *partition*:

$$\begin{aligned} & part1 \quad : \quad (fun\ partition)^{\smile} \circ fun\ \mathsf{inj}_2 \sqsubseteq permute \circ fun\ join \circ okl\ ¿ \\ & part2 \quad : \quad (\lambda b \to isInj_1\ (partition\ b)) \subseteq nil, \end{aligned}$$

where $isInj_1\ x \ = \ x \equiv \mathsf{inj}_1\ tt$. Therefore, (3) can be refined to

$$foldT\ ((fun\ partition)^{\smile} \circ fun\ \mathsf{inj}_2)\ (\lambda b \to isInj_1\ (partition\ b)),$$

as is done in step 6, using $foldT\text{-}monotonic$.

Now the specification has been transformed into a fold followed by the converse of another fold, using a functional argument *partition*. The following lemma, applied

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

in step 7, allows one to refine it to a functional unfold:

$$\begin{aligned}&foldT\text{-}to\text{-}unfoldt \;:\; \{A\ B : Set\} \to (f : B \to \top \uplus (A \times B \times B)) \to \\ &\quad (wf : well\text{-}found\ (\varepsilon\text{-}TreeF \circ fun\ f)) \to \\ &\qquad (foldT\ ((fun\ f)^{\smile} \circ (fun\ \mathsf{inj}_2))\ (\lambda b \to isInj\ (f\ b)))^{\smile} \sqsupseteq fun\ (unfoldt\ f\ wf).\end{aligned}$$

The proof of *foldT-to-unfoldt*, which merely states that the right-hand side *fun* (*unfoldt f wf*) always returns a result allowed by the left-hand side, is not hard and is omitted here.

With the theories developed in Section 6.4, we do not have to prove the well-foundedness for *partition* from scratch. We notice that the sub-lists returned by *partition* must have lengths strictly smaller than the input list:

$$partition \sqsubseteq\!< \;:\; fun\ length \circ (\varepsilon\text{-}TreeF \circ fun\ partition) \circ (fun\ length)^{\smile} \sqsubseteq \_<\_.$$

Given that $\_<\_$ is well founded, the well-foundedness of *fun length* $\circ$ (ε-*TreeF* $\circ$ *fun partition*) $\circ$ (*fun length*)$^{\smile}$ can be established by *acc*-$\sqsubseteq$, from which we may prove, by *acc-fRf*$^{\smile}$, the well-foundedness of ε-*TreeF* $\circ$ *fun partition*:

$$\begin{aligned}&partition\text{-}wf \;:\; well\text{-}found\ (\varepsilon\text{-}TreeF \circ fun\ partition) \\ &partition\text{-}wf\ xs \;=\; acc\text{-}fRf^{\smile}\ xs \\ &\qquad (acc\text{-}\sqsubseteq\ partition \sqsubseteq\!<\ (length\ xs)\ \ (\mathbb{N}<\text{-}wf\ (length\ xs))).\end{aligned}$$

The folding and unfolding phases are merged by the lemma *fun*$\circ$-$\sqsupseteq$ in step 8. We have thus derived quicksort expressed as a hylomorphism. The complete derivation takes about 500 lines of code.

7 Conclusion and related work

We have shown how to encode relational program derivation in a dependently typed language. Derivation is carried out in the host language, the correctness being guaranteed by the type system. Various concepts often used in relational program derivation, including relational folds, division and minimum, can be modelled with dependent types. We have presented several non-trivial derivations, including an optimisation problem, and a relational derivation of quicksort, where well-founded recursion is used to prove the termination of the unfolding phase in a hylomorphism.

There is plenty of scope for future work. As many readers and the referees pointed out, while we encode the derivations in a dependently typed programming language, the programs we derive remain non-dependently typed. It would be interesting to see whether indexed datatypes go well with point-free programs so that partiality of functions, currently represented using coreflexive relations in Section 5, can be encoded in the datatype. All the program derivations we have dealt with handle first-order data. It also remains to see whether it is sufficient for most cases.

McKinna & Burstall's (1993) paper on 'deliverables' is an early example of machine-checked program + proof construction (using Pollack's LEGO). In their terminology *sort-der* would be a deliverable – an element of a dependent Σ-type, pairing up a function and a proof of correctness. In the Coq tradition program extraction has been used already from Paulin-Mohring's (1989) early paper for

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

the impressive four-colour theorem development, including the development of a verified compiler (Leroy 2006). Our contribution is more modest – we aim at formally checked but still readable algebra-of-programming style derivations.

The concept of inductive families (Dybjer 1994), especially the identity type ($\_\equiv\_$), is central to the Agda system and to our derivations. A recent development of relations in dependent type theory was carried out by Gonzalía (2006, Chapter 5). The advances in Agda's notation and support for hidden arguments between that derivation and our work is striking.

There has been a trend in recent years to bridge the gap between dependent types and practical programming. Projects along this line include Cayenne (Augustsson 1998), Coq (Coq Development Team 2006), Dependent ML (Xi 2007), Agda (Norell 2007), Ωmega (Sheard 2007), Epigram (McBride & McKinna 2004) and the GADT extension (Cheney & Hinze 2003) to Haskell. It is believed that dependent types have an important role in the next generation of programming languages (Sweeney 2006).

The concept of well-foundedness has long been developed in recursive function theory. In computing science, Floyd (1967) proposed the use of well-ordering in programming language semantics (See Cousot 1990 for a tutorial). More recently, Megacz (2007) gave a nice survey of several alternative approaches to code terminating programs in a dependently typed language with inductive and coinductive types.

Acknowledgments

We are grateful to Nils Anders Danielsson and Peter Dybjer for giving valuable technical and presentational suggestions and pointing us to a number of useful references. We would also like to thank the anonymous referees for suggesting plenty of improvements on the paper. This project was inspired by some initial experiments conducted by Max Schäfer.

References

Agda Team, The. (2007) The Agda wiki [online]. Available at: `http://wiki.portal.chalmers.se/agda/` (Accessed 3 July 2009).

Augustsson, L. (1998) Cayenne – a language with dependent types. In *ACM SIGPLAN International Conference on Functional Programming*, Felleisen, M., Hudak, P. & Queinnec, C. (eds), ACM Press, pp. 239–250.

Augustsson, L. (1999) Equality proofs in Cayenne [online]. Available at: `http://www.cs.chalmers.se/~augustss/cayenne/eqproof.ps` (Accessed 3 July 2009).

Backhouse, R. C. (2002) Galois connections and fixed point calculus. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Backhouse, R. C., Crole, R. & Gibbons, J. (eds), LNCS, no. 2297. Springer, pp. 89–148.

Backhouse, R. C. & Hoogendijk, P. F. (1992) Elements of a relational theory of datatypes. In *Formal Program Development*, Möller, B., Partsch, H. A. & Schuman, S. A. (eds), LNCS, no. 755. Springer, pp. 7–42.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

Backhouse, R. C., de Bruin, P. J., Malcolm, G., Voermans, E. & van der Woude, J. (1991) Relational catamorphisms. In *IFIP TC2/WG2.1 Working Conference on Constructing Programs*, Möller, B. (eds), Elsevier, pp. 287–318.

Bird, R. S. (1989a) Algebraic identities for program calculation, *Comp. J.*, **32** (2): 122–126.

Bird, R. S. (1989b) Lectures on constructive functional programming. In *Constructive Methods in Computing Science*, Broy, M. (ed), NATO ASI Series F, vol. 55. Springer, pp. 151–216.

Bird, R. S. (1996) Functional algorithm design, *Sci. Comp. Program.*, **26**: 15–31.

Bird, R. S. & de Moor, O. (1997) *Algebra of Programming*, International Series in Computer Science. Prentice Hall.

Bove, A. & Capretta, V. (2005) Modelling general recursion in type theory, *Math. Struct. Comp. Sci.*, **15** (4): 671–708.

Brady, E., McBride, C. & McKinna, J. (2003) Inductive families need not store their indices. In *Types for Proofs and Programs*, Berardi, S., Coppo, M. & Damiani, F. (eds), LNCS, vol. 3085. Springer, pp. 115–129.

Burstall, R. M. (1969) Proving properties of programs by structural induction, *Comp. J.*, **12** (1): 41–48.

Cheney, J. & Hinze, R. (2003) *First-Class Phantom Types*, Technical Report TR2003-1901. Cornell University.

Coq Development Team, The. (2006) *The Coq Proof Assistant Reference Manual*. LogiCal Project.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001) *Introduction to Algorithms*. MIT Press.

Cousot, P. (1990) Method and logics for proving programs. In *Formal Models and Semantics*, van Leeuwen, J. (ed), Handbook of Theoretical Computer Science, vol. B. Elsevier, pp. 843–993.

Danielsson, N. A., Norell, U., Mu, S-C., Bronson, S., Doel, D., Jansson, P. & Chen, L-T. (2009) The Agda standard library [online]. Available at: `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Libraries.StandardLibrary` (Accessed 3 July 2009).

Doornbos, H. (1996) *Reductivity Arguments and Program Construction*, PhD thesis. Eindhoven University of Technology.

Doornbos, H. & Backhouse, R. C. (1995) Induction and recursion on datatypes. In *Mathematics of Program Construction 1995*, LNCS, vol. 947. Springer, pp. 242–256.

Doornbos, H. & Backhouse, R. C. (1996) Reductivity, *Sci. Comp. Program.*, **26**: 217–236.

Dybjer, P. (1994) Inductive families, *Formal Aspects Comput.*, **6** (4): 440–465.

Dybjer, P. & Setzer, A. (1999) A finite axiomatization of inductive-recursive definitions. In *TLCA'99*, Girard, J.-Y. (ed), LNCS, vol. 1581. Springer, pp. 129–146.

Floyd, R. W. (1967) Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, Schwartz, J. T. (ed), Proceedings of Symposia in Applied Mathematics, vol. 19. American Mathematical Society, pp. 19–32.

Goguen, H., McBride, C. & McKinna, J. (2006) Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, Futatsugi, K., Jouannaud, J.-P. & Meseguer, J. (eds), LNCS, vol. 4060. Springer, pp 521–540.

Gonzalía, C. (2006) *Relations in Dependent Type Theory*, PhD thesis. Chalmers University of Technology.

Gries, D. (1989) The maximum-segment-sum problem. In *Formal Development Programs and Proofs*, Dijkstra, E. W. (ed), University of Texas at Austin Year of Programming Series. Addison-Wesley, pp. 33–36.

Harper, R. & Pollack, R. (1991) Type checking with universes, *Theoret. Comp. Sci.*, **89** (1): 107–136.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press

Leroy, X. (2006) Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *The 33th Symposium on Principles of Programming Languages*. ACM Press, pp. 42–54.

Magnusson, L. & Nordström, B. (1994) The ALF proof editor and its proof engine. In *Proceedings of the International Workshop on Types for Proofs and Programs*. Springer, LNCS 806, pp. 213–237.

McBride, C. & McKinna, J. (2004) The view from the left, *J. Funct. Program.*, **14** (1): 69–111.

McKinna, J. & Burstall, R. M. (1993) Deliverables: A categorical approach to program development in type theory. In *International Symposium on Mathematical Foundations of Computer Science*, Borzyszkowski, A. M. & Sokolowski, S. (eds), Springer, LNCS no. 711, pp. 32–67.

Megacz, A. (2007) A coinductive monad for prop-bounded recursion. In *Proceedings of the ACM Workshop Programming Languages meets Program Verification*, Stump, A. & Xi, H. (eds). ACM Press, pp. 11–20.

Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes, and barbed wire. In *ACM Conference on Functional Programming Languages and Computer Architecture*, Hughes, J. (ed), Springer-Verlag, pp. 124–144.

Mu, S-C. (2008) Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths. In *ACM SIGPLAN 2008 Symposium on Partial Evaluation and Program Manipulation*. ACM Press, pp. 31–39.

Mu, S.-C. & Bird, R. S. (2003) Theory and applications of inverting functions as folds, *Sci. Comp. Program.*, **51**: 87–116.

Mu, S-C., Ko, H-S. & Jansson, P. (2008a) Algebra of programming using dependent types. In *Mathematics of Program Construction 2008*, Audebaud, P. & Paulin-Mohring, C. (eds), Springer, LNCS 5133, pp. 268–283.

Mu, S-C., Ko, H-S. & Jansson, P. (2008b) AoPA: Algebra of programming in Agda [online]. Available at: `http://www.iis.sinica.edu.tw/~scm/2008/aopa/` (Accessed 3 July 2009).

Nordström, B. (1988) Terminating general recursion, *BIT Numer. Math.*, **28** (3): 605–619.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*, PhD thesis. Chalmers University of Technology.

Paulin-Mohring, C. (1989) Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Symposium on Principles of Programming Languages*. ACM.

Sheard, T. & Linger, N. (2008) Central European Functional Programming School. In *Programming in Ωmega*, Horváth, Z., Plasmeijer, R., Soós, A. & Zsók, V. (eds), Springer-Verlag, LNCS no. 5161, pp. 158–227.

Sweeney, T. (2006) The next mainstream programming language: A game developer's perspective. In *Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 11–13.

Tarski, A. (1955) A lattice-theoretic fixpoint theorem and its applications, *Pacific. Math.*, **5**: 285–309.

Wadler, P. (1990) Deforestation: Transforming programs to eliminate trees, *Theoret. Comp. Sci.*, **73**: 231–248.

Xi, H. (2007) Dependent ML: An approach to practical programming with dependent types, *J. Funct. Program.*, **17** (2): 215–286.

https://doi.org/10.1017/S0956796809007345 Published online by Cambridge University Press