Algebra of Programming using Dependent Types

Shin-Cheng Mu[1], Hsiang-Shang Ko[2], and Patrik Jansson[3]

[1] Institute of Information Science, Academia Sinica, Taiwan
[2] Department of Computer Science and Information Engineering
National Taiwan University, Taiwan
[3] Department of Computer Science and Engineering
Chalmers University of Technology & University of Gothenburg, Sweden

Abstract. Dependent type theory is rich enough to express that a program satisfies an input/output relational specification, but it could be hard to construct the proof term. On the other hand, squiggolists know very well how to show that one relation is included in another by algebraic reasoning. We demonstrate how to encode functional and relational derivations in a dependently typed programming language. A program is coupled with an algebraic derivation from a specification, whose correctness is guaranteed by the type system.

1 Introduction

Program derivation is the technique of successively applying formal rules to a specification to obtain a program that is correct by construction. On the other hand, modern programming languages deploy expressive type systems to guarantee compiler-verifiable properties. There has been a trend to explore the expressiveness of dependent types, which opens a whole new world of type-level programming techniques. As Altenkirch et al. [1] put it, dependently typed programs are, "by their nature, proof carrying code." This paper aims to illustrate their comment by showing, in the dependently typed language Agda [17], that programs can actually carry their derivations.

As a teaser, Fig. 1 shows a derivation of a sorting algorithm in progress. The type of *sort-der* is a proposition that there exists a program of type $[\mathit{Val}] \rightarrow [\mathit{Val}]$ that is contained in $\mathit{ordered}? \circ \mathit{permute}$, a relation mapping a list to one of its ordered permutations. The proof proceeds by derivation from the specification towards the algorithm. The first step exploits monotonicity of $\circ$ and that *permute* can be expressed as a fold. The second step makes use of relational fold fusion. The shaded areas denote *interaction points* — fragments of (proof) code to be completed. 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. It is an executable program that is backed by the type system to meet the specification.

We have developed a library for functional and relational program derivation, with convenient notation for algebraic reasoning. Our work aims to be a co-operation between the *squiggolists* and dependently-typed programmers that

```
sort-der  :  ∃ ([Val] → [Val]) (\f → ordered? ∘ permute ⊒ fun f)
sort-der  =  exists _(     ordered? ∘ permute
                        ⊒⟨  (\vs → ▪-monotonic ordered? (permute-is-fold vs))  ⟩
                            ordered? ∘ foldR combine nil
                        ⊒⟨   {foldR-fusion ordered? ins-step ins-base}0   ⟩
                            {  }1 )

isort  :  [Val] → [Val]
isort  =  witness sort-der
```

Fig. 1. A derivation of a sorting algorithm in progress (see Sect. 4 for the details).

may benefit both sides. On the one hand, a number of tools for program transformation [22, 11, 24] have been developed but few of them have been put into much use. Being able to express derivation *within* the programming language encourages its use and serves as documentation. This paper is a case study of using the Curry-Howard isomorphism which the squiggolists may appreciate: specification of the program is expressed in their types, whose proofs (derivations) are given as programs and checked by the type system. On the other hand, it is known among dependently-typed programmers that the expressiveness of dependent types is far beyond proving that *reverse* preserves the length of its input. We can reason about the full input/output specification, for example, that *fast-reverse* is pointwise equal to the quadratic-time *reverse*, or that insertion sort implements a relational specification of sort. The reason this is rarely done is probably because it appears difficult to construct the proof terms. The method we propose is to develop the proof by algebraic reasoning within Agda.

In Sect. 2 we give an introduction to the part of Agda we use. We present our encoding of relations and their operations in Sect. 3, which prepares us to discuss our primary example in Sect. 4 and conclude with related work in Sect. 5.

2 A Crash Course on 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 on Agda, focusing on the aspects we need. For a detailed documentation, the reader is referred to Norell [17] and the Agda wiki [21].

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

```
id : (A : Set) → A → A
id A a  =  a,
```

where *Set* is the kind of 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 when 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 below:

$$\begin{aligned}&id : \{A : Set\} \to A \to A\\ &id\ a \ = \ a,\end{aligned}$$

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

As an example of a datatype definition, cons-lists can be defined by:

$$\begin{aligned}&\textbf{data}\ [\_]\ (A : Set) : Set\ \textbf{where}\\ &\quad [\,]\quad\ : [A]\\ &\quad \_::\_\ \ : A \to [A] \to [A].\end{aligned}$$

In Agda's notation for dist-fix definitions, an underline denotes a location for a parameter. The type constructor $[\_]$ 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 $[\,]$.

2.1 First-Order 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. Fig. 2 shows an encoding of first-order intuitionistic logic in Agda. Falsity is represented by $\bot$, a type with no constructors and therefore no inhabitants. Truth, on the other hand, can be represented by the type $\top$, having one unique term — a record with no fields. Disjunction is represented by disjoint sum, while conjunction is denoted by product as usual: a proof of $P \uplus Q$ can be deducted either from a proof of P or a proof of Q, while a proof of $P \times Q$ consists of proofs of both.

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. The quantifier $\forall$ is encoded as a dependent function which, given any $x : A$, must produce a proof of $P\,x$. Agda provides a short hand $forall\,x \to P\,x$ in place of $(x : A) \to P\,x$ when A can be inferred. To prove the proposition $\exists\,A\,P$, where P is a predicate on terms of type A, one has to provide a witness $w : A$ and a proof of $P\,w$. Given a term of type $\exists\,A\,P$, the two functions *witness* and *proof* extract the witness and the proof, respectively.

```
data ⊥ : Set where

record ⊤ : Set where

data _⊎_ (P Q : Set) : Set where
  inj₁ : P → P ⊎ Q
  inj₂ : Q → P ⊎ Q

data _×_ (P Q : Set) : Set where
  _,_ : P → Q → P × Q
```

```
data ∃ (A : Set) (P : A → Set) : Set
 where
  exists : (w : A) → P w → ∃ A P

witness : {A : Set}{P : A → Set} →
        ∃ A P → A
witness (exists w p)  =  w

proof : {A : Set}{P : A → Set} →
        (x : ∃ A P) → P (witness x)
proof (exists w p)  =  p
```

Fig. 2. An encoding of first-order intuitionistic logic in Agda.

2.2 Identity Type

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

```
data _≡_ {A : Set}(x : A) : A → Set where
    ≡-refl  :  x ≡ x.
```

Agda has relaxed lexical rules allowing Unicode characters in identifiers. Therefore, ≡-*refl* (without space) is a valid name. Since the only constructor ≡-*refl* is of type $x \equiv x$, being able to type-check a term with type $x \equiv y$ means that the type checker is able to deduce that x and y are indeed equal[4].

For the rest of the paper, we will exploit Unicode characters to give telling names to constructors, arguments, and lemmas. For example, if a variable is a proof of $y \equiv z$, we may name it *y≡z* (without space).

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

```
≡-sym : {A : Set}{x y : A} →  x ≡ y  →  y ≡ x
≡-sym ≡-refl  =  ≡-refl,

≡-trans : {A : Set}{x y z : A} →  x ≡ y  →  y ≡ z  →  x ≡ z
≡-trans ≡-refl y≡z  =  y≡z.
```

The type of the first explicit parameter in the type signature of ≡-*sym* is $x \equiv y$, while the constructor ≡-*refl* in the pattern has type $x \equiv x$. When the type checker tries to unify them, x is unified with y. Therefore, when we need to return a term of type $y \equiv x$ on the right-hand side, we can simply return ≡-*refl*. The situation with ≡-*trans* is similar. Firstly, x is unified with y, therefore the parameter *y≡z*, having type $y \equiv z$, can also be seen as having type $x \equiv z$ and

[4] Agda assume uniqueness of identity proofs (but not proof irrelevance).

```
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. 3. Combinators for preorder reasoning.

be returned. In general, pattern matching and inductive families (such as _≡_) is a very powerful combination.

The interactive feature of Agda[5] is helpful for constructing the proof terms. 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.

The lemma ≡-*subst* states that Leibniz equality holds: if $x \equiv y$, they are interchangeable in all contexts. Given a context f and a proof that $x \equiv y$, the congruence lemma ≡-*cong* returns a proof that $f\,x \equiv f\,y$.

```
≡-subst : {A : Set}(P : A → Set){x y : A} →  x ≡ y  → P x → P y
≡-subst P ≡-refl Px   =   Px,

≡-cong : {A B : Set}(f : A → B){x y : A} →  x ≡ y  →  f x ≡ f y
≡-cong f ≡-refl   =   ≡-refl.
```

2.3 Preorder Reasoning

To prove a proposition $e_1 \equiv e_2$ is to construct a term having that type. One can do so by the operators defined in the previous section. It can be very tedious, however, when the expressions involved get complicated. Luckily, for any binary relation _∼_ that is reflexive and transitive (that is, for which one can construct terms ∼-*refl* and ∼-*trans* having the types as described in the previous section), we can induce a set of combinators, shown in Fig. 3, which allows one to construct a term of type $e_1 \sim e_n$ in algebraic style. These combinators are implemented in Agda by Norell [17] and improved by Danielsson in the Standard Library of Agda [21]. Augustsson [3] has proposed a similar syntax for equality reasoning, with automatic inference of congruences.

[5] Agda has an Emacs mode and a command line interpreter interface.

$$
\begin{array}{l}
\textit{foldr-fusion} : \{A\ B\ C : \textit{Set}\} \to (h : B \to C) \to \{f : A \to B \to B\} \to \\
\quad \{g : A \to C \to C\} \to \{z : B\} \to (\textit{push} : \textit{forall}\ a \to h \cdot f\ a \doteq g\ a \cdot h) \to \\
\qquad h \cdot \textit{foldr}\ f\ z \doteq \textit{foldr}\ g\ (h\ z) \\
\textit{foldr-fusion}\ h\ \{f\}\ \{g\}\ \{z\}\ \_ \quad [\,] \qquad = \ \equiv\text{-}\textit{refl} \\
\textit{foldr-fusion}\ h\ \{f\}\ \{g\}\ \{z\}\ \textit{push}\ (a :: \textit{as}) = \\
\quad \textbf{let}\ \textit{ih} = \textit{foldr-fusion}\ h\ \textit{push}\ \textit{as}\ \textbf{in} \\
\qquad h\ (\textit{foldr}\ f\ z\ (a :: \textit{as})) \\
\quad \equiv\langle\ \ \equiv\text{-}\textit{refl}\ \ \rangle \\
\qquad h\ (f\ a\ (\textit{foldr}\ f\ z\ \textit{as})) \\
\quad \equiv\langle\ \ \textit{push}\ a\ (\textit{foldr}\ f\ z\ \textit{as})\ \ \rangle \\
\qquad g\ a\ (h\ (\textit{foldr}\ f\ z\ \textit{as})) \\
\quad \equiv\langle\ \ \equiv\text{-}\textit{cong}\ (g\ a)\ \textit{ih}\ \ \rangle \\
\qquad g\ a\ (\textit{foldr}\ g\ (h\ z)\ \textit{as}) \\
\quad \equiv\langle\ \ \equiv\text{-}\textit{refl}\ \ \rangle \\
\qquad \textit{foldr}\ g\ (h\ z)\ (a :: \textit{as}) \\
\quad \equiv\Box
\end{array}
$$

Fig. 4. Proving the fusion theorem for *foldr*.

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

$$
\begin{array}{l}
\qquad e_1 \\
\sim\langle\ \ \textit{reason}_1\ \ \rangle \\
\qquad\ \vdots \\
\qquad e_{n-1} \\
\sim\langle\ \ \textit{reason}_{n-1}\ \ \rangle \\
\qquad e_n \\
\sim\Box
\end{array}
$$

should be bracketed as $e_1 \sim\langle \textit{reason}_1 \rangle \ldots (e_{n-1} \sim\langle \textit{reason}_{n-1} \rangle\, (e_n {\sim}\Box))$. Each occurrence of $\_{\sim}\langle\_\rangle\_$ takes three arguments: e_i on the left, $\textit{reason}_i$ (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 $e_n \sim e_n$.

2.4 Functional Derivation

The ingredients we have prepared so far already allow us to perform some functional program derivation. For brevity, however, we introduce an equivalence relation on functions:

$$
\begin{array}{l}
\_\doteq\_ : \{A\ B : \textit{Set}\} \to (A \to B) \to (A \to B) \to \textit{Set} \\
f \doteq g \ = \ \textit{forall}\ a \to f\ a \equiv g\ a.
\end{array}
$$

Since $\_\doteq\_$ can be shown to be reflexive and transitive, it also induces a set of preorder reasoning operators. Fig. 4 shows a proof of the foldr fusion theorem. The

$$
\begin{array}{l}
\mathit{scanr\text{-}der} : \{A\,B : \mathit{Set}\} \to (f : A \to B \to B) \to (e : B) \to \\
\qquad \exists\,([A] \to \mathit{List}^{+}\,B)\,(\backslash \mathit{prog} \to \mathit{map}^{+}\,(\mathit{foldr}\,f\,e) \cdot \mathit{tails} \doteq \mathit{prog}) \\
\mathit{scanr\text{-}der}\,f\,e \;=\; \mathit{exists}\,\_(\quad \mathit{map}^{+}\,(\mathit{foldr}\,f\,e)\cdot \mathit{tails} \\
\qquad\qquad\qquad\qquad\quad \doteq\langle\quad \mathit{foldr\text{-}fusion}\,(\mathit{map}^{+}\,(\mathit{foldr}\,f\,e))\,(\mathit{push\text{-}map\text{-}til}\,f)\quad \rangle \\
\qquad\qquad\qquad\qquad\qquad \mathit{foldr}\,(\mathit{sc}\,f)\,[e]^{+} \\
\qquad\qquad\qquad\qquad\quad \doteq\Box) \\
\quad \textbf{where}\,\mathit{sc} : \{A\,B : \mathit{Set}\} \to (A \to B \to B) \to A \to \mathit{List}^{+}\,B \to \mathit{List}^{+}\,B \\
\qquad\quad \mathit{sc}\,f\;a\,[b]^{+} \qquad\;\; = f\;a\;b ::^{+} [b]^{+} \\
\qquad\quad \mathit{sc}\,f\;a\,(b ::^{+} bs) = f\;a\;b ::^{+} b ::^{+} bs \\
\qquad\quad \mathit{push\text{-}map\text{-}til} : \{A\,B : \mathit{Set}\} \to (f : A \to B \to B) \to \{e : B\} \to (a : A) \to \\
\qquad\qquad\quad \mathit{map}^{+}\,(\mathit{foldr}\,f\;e)\cdot \mathit{til}\;a \doteq \mathit{sc}\,f\;a \cdot \mathit{map}^{+}\,(\mathit{foldr}\,f\;e) \\
\qquad\quad \mathit{push\text{-}map\text{-}til}\;f\;a\,[xs]^{+} \qquad\quad = \equiv\text{-}\mathit{refl} \\
\qquad\quad \mathit{push\text{-}map\text{-}til}\;f\;a\,(xs ::^{+} xss) = \equiv\text{-}\mathit{refl}
\end{array}
$$

Fig. 5. Derivation of *scanr*. The constructors $\_::^{+}\_$ and $[\_]^{+}$ build non-empty lists, while $\mathit{tails} = \mathit{foldr}\;\mathit{til}\,[[]]^{+}$, where $\mathit{til}\;a\,[xs]^{+} = (a::xs)::^{+}[xs]^{+}$; $\mathit{til}\;a\,(xs ::^{+} xss) = (a::xs) ::^{+} xs ::^{+} xss$.

steps using $\equiv$-*refl* are simple equivalences which Agda can prove by expanding the definitions. The inductive hypothesis *ih* is established by a recursive call. Agda ensures that proofs by induction are well-founded. Fig. 5 derives *scanr* from its specification $\mathit{map}^{+}\,(\mathit{foldr}\,f\;e)\cdot\mathit{tails}$, 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 $\mathit{scanr} = \mathit{witness}\;\mathit{scanr\text{-}der}$, while $\mathit{scanr\text{-}pf} = \mathit{proof}\;\mathit{scanr\text{-}der}$ is a proof that can be used elsewhere. Notice that the first argument to *exists* is left implicit. Agda is able to infer the witness because it is syntactically presented in the derivation.

We have reproduced a complete derivation for the maximum segment sum problem. The derivation proceeds in the standard manner [6], transforming the specification to $\mathit{max}\cdot\mathit{map}\,(\mathit{foldr}\,\_\otimes\_\,0)\cdot\mathit{tails}$ for some $\_\otimes\_$, and exploiting *scanr-pf* to convert it to a *scanr*. The main derivation is about 220 lines long, plus 400 lines of library code proving properties about lists and 100 lines for properties about integers. The code is available online [16].

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.

3 Relational Derivation

During the 90’s there was a trend in the program derivation community to move from functions to relations. For completeness, we give a quick introduction to relations in this section. The reader is referred to Backhouse et al. [4] and Backhouse and Hoogendijk [5] for a more rigorous treatment. Bird and de Moor [8] present program derivation from a more abstract, category-theoretical point of view, with many illustrative examples of program derivation.

A relation R to B from A, denoted by $R : B \leftarrow A$, is usually understood as a subset of the set of pairs $A \times B$.[6] A function f is seen as a special case where $(a, b) \in f$ and $(a, b') \in f$ implies $b = b'$. The use of relations allows non-determinism in the specification. Derivation proceeds by inclusion as well as equality: in each step, the specification may be refined to a more deterministic subset, often all the way until we reach a function.

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

The fold remains an important construct in the relational setting. While *foldr* takes a step function of type $A \to B \to B$ and a base case of type B, its relational counterpart, which we denote by *foldR*, takes an uncurried relation $R : B \leftarrow (A \times B)$, while the base cases, being non-deterministic, are recorded in a set $s : \mathsf{P}B$.[7] The relational fold can be defined in terms of functional fold:

$$\begin{aligned}&\mathit{foldR}\; R\; s \;:\; B \leftarrow [A]\\&\mathit{foldR}\; R\; s = \in \circ\, \mathit{foldr}\; \Lambda(R \circ (\mathit{id} \times \in))\; s.\end{aligned}$$

We will see in the next few sections how these concepts can be modelled in Agda.

3.1 Modelling Relations

A possibly infinite subset (of A) could be represented by its membership function of type $\mathsf{P}A = A \to \mathit{Bool}$. With dependent types, we can also represent the membership judgement at type level:

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

A set $s : \mathsf{P}A$ is a function mapping $a : A$ to a type, which encodes a logic formula determining its membership. Agda maintains a hierarchy of universes, where *Set* denotes the universe of types, *Set*1 denotes the universe of *Set* and all types declared as being in *Set*1, etc. Since $s : \mathsf{P}A$ is a function yielding a *Set*, $\mathsf{P}A$ is in the universe *Set*1. The function *singleton* creates singleton sets:

$$\begin{aligned}&\mathit{singleton} : \{A : \mathit{Set}\} \to A \to \mathsf{P}A\\&\mathit{singleton}\; a = \backslash a' \to\;\; a \equiv a'.\end{aligned}$$

Set union and inclusion, for example, are naturally encoded by disjunction and implication:

[6] Notations used in the beginning of this section, for example $\times$, $\in$, and set comprehension, refer to their usual set-theoretical definitions. We will talk about how they can be represented in Agda in the next few subsections.

[7] Isomorphically, the base case can be represented by a relation $B \leftarrow \top$.

$$\_\cup\_ : \{A : Set\} \to \mathsf{P}A \to \mathsf{P}A \to \mathsf{P}A \qquad \_\subseteq\_ : \{A : Set\} \to \mathsf{P}A \to \mathsf{P}A \to Set$$
$$r \cup s \;=\; \backslash\, a \to \;\; r\; a \uplus s\; a, \qquad\qquad r \subseteq s \;=\; forall\; a \to \;\; r\; a \to s\; a.$$

A term of type $r \subseteq s$ is a function which, given an a and a proof that a is in r, produces a proof that a is in s.

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

$$\_\leftarrow\_ \;:\; Set \to Set \to Set1$$
$$B \leftarrow A \;=\; A \to B \to Set.$$

One of the advantages is that relations and set-valued functions are unified. The Λ operator, for example, is simply the identity function at the term-level:

$$\Lambda \quad : \{A\,B : Set\} \to (B \leftarrow A) \to (A \to \mathsf{P}B)$$
$$\Lambda\, R = R.$$

A function can be converted to a relation:

$$fun : \{A\,B : Set\} \to (A \to B) \to (B \leftarrow A)$$
$$fun\, f\;\; a\; b \;=\; f\; a \equiv b.$$

The identity relation, for example, is denoted $id_R : \{A : Set\} \to (A \leftarrow A)$ and defined by $id_R = fun\; id$.

Relational composition could be defined by $R \circ S = \exists\, B\, (\backslash b \to (S\; a\; b \times R\; b\; c))$. For reasons that will be clear in the next section, we split the definition into two parts, shown in Fig. 6. The operator $\_\bullet\_$ applies a relation $R : B \leftarrow A$ to a set $\mathsf{P}A$, yielding another set $\mathsf{P}B$. Composition $\_\circ\_$ is then defined using $\_\bullet\_$.

Complication arises when we try to represent $\in$. Recall that $\in$ maps $\mathsf{P}A$ to A. However, the second argument to $\_\leftarrow\_$ must be in Set, while $\mathsf{P}A$ is in $Set1$! At present, we have no choice but to declare another type of arrows that accepts $Set1$-sorted inputs:

$$\_\leftarrow_1\_ \;:\; Set \to Set1 \to Set1$$
$$B \leftarrow_1 PA \;=\; PA \to B \to Set.$$

It means we need several alternatives of relational composition that differ only in their types. Fig. 6 shows $\_\circ_1\_$ and $\_{}_1\!\circ\_$ for example. Such inconvenience may be resolved if Agda introduces *universe polymorphism*, a feature on the wish-list at the time of writing. Also summarised in Fig. 6 are $\_\breve{}$ for relational converse, and $\_\times_1\_$, a higher-kinded variation of the product functor.

3.2 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:

$$\_\sqsubseteq\_ : \{A\,B : Set\} \to (B \leftarrow A) \to (B \leftarrow A) \to Set$$
$$R \sqsubseteq S = forall\; a \to R\; a \subseteq S\; a,$$

$$
\begin{array}{ll}
\_\breve{} : \{A\; B : \mathit{Set}\} \to (A \leftarrow B) \to (B \leftarrow A) & \in\; : \{A : \mathit{Set}\} \to (A \leftarrow_1 \mathsf{P}A) \\
R\breve{} = \backslash\, a\; b \;\to\; R\; b\; a & \in\; = \; \backslash pa\; a \to pa\; a
\end{array}
$$

$$
\begin{array}{l}
\_{\scriptstyle\blacksquare}\_ : \{A\; B : \mathit{Set}\} \to (B \leftarrow A) \to \mathsf{P}A \to \mathsf{P}B \\
R \,{\scriptstyle\blacksquare}\, s = \backslash\, b \;\to\; \exists\, A\, (\backslash a \to (s\; a \times R\; a\; b))
\end{array}
$$

$$
\begin{array}{l}
\_\circ\_ : \{A\; B\; C : \mathit{Set}\} \to (C \leftarrow B) \to (B \leftarrow A) \to (C \leftarrow A) \\
(R \circ S)\; a \;=\; R \,{\scriptstyle\blacksquare}\, (S\; a)
\end{array}
$$

$$
\begin{array}{l}
\_\circ_1\_ : \{A : \mathit{Set1}\}\{B\; C : \mathit{Set}\} \to (C \leftarrow B) \to (B \leftarrow_1 A) \to (C \leftarrow A) \\
(R \circ_1 S)\; a \;=\; R \,{\scriptstyle\blacksquare}\, (S\; a)
\end{array}
$$

$$
\begin{array}{l}
\_{}_1\!\circ\_ : \{A\; B\; C : \mathit{Set}\} \to (C \leftarrow_1 \mathsf{P}B) \to (B \leftarrow A) \to (C \leftarrow A) \\
(R\; {}_1\!\circ S)\; a \;=\; R\, (S\; a)
\end{array}
$$

$$
\begin{array}{l}
\_\times_1\_ : \{A\; B : \mathit{Set}\}\{PC : \mathit{Set1}\}\{D : \mathit{Set}\} \to \\
\qquad (B \leftarrow A) \to (D \leftarrow_1 PC) \to ((B \times D) \leftarrow_1 (A \times_1 PC)) \\
(R \times_1 S)\, (a ,_1 pc)\, (b, d) \;=\; R\; a\; b \times S\; pc\; d
\end{array}
$$

Fig. 6. Some operators on $\_\leftarrow\_$ and $\_\leftarrow_1\_$ relations, including composition, membership and product. In this paper, $\_\times_1\_$ is overloaded for the type of pairs whose right component is in *Set*1 ($\_,_1\_$ being the data constructor), and its functor action on relations (defined in this figure).

which expands to $\mathit{forall}\; a \to \mathit{forall}\; b \to R\; a\; b \to S\; a\; b$. Conversely, $R \sqsupseteq S = S \sqsubseteq R$. Both $\sqsubseteq$ and $\sqsupseteq$ can be shown to be reflexive and transitive. Therefore, we can use them for preorder reasoning.

In hand-written derivation, the monotonicity of $\_\circ\_$ (that is, $S \sqsubseteq T$ implies $R \circ S \sqsubseteq R \circ T$) is often used without being explicitly stated. In our Agda encoding where there are many versions of composition, it appears that we need one monotonicity lemma for each of them. Luckily, since those alternatives of composition are all defined in terms of $\_{\scriptstyle\blacksquare}\_$, it is enough to model monotonicity for $\_{\scriptstyle\blacksquare}\_$ only:

$$
\begin{array}{l}
{\scriptstyle\blacksquare}\text{-}\mathit{monotonic} : \{A\; B : \mathit{Set}\} \to (R : B \leftarrow A) \to \{s\; t : \mathsf{P}A\} \to \\
\qquad\qquad\qquad s \subseteq t \to R \,{\scriptstyle\blacksquare}\, s \subseteq R \,{\scriptstyle\blacksquare}\, t \\
{\scriptstyle\blacksquare}\text{-}\mathit{monotonic}\; R\; s{\subseteq}t\; b\; (\mathit{exists}\; a_1\; (a_1{\in}s,\; bRa_1)) = \\
\qquad \mathit{exists}\; a_1\; (s{\subseteq}t\; a_1\; a_1{\in}s,\; bRa_1).
\end{array}
$$

To refine $R \circ S \circ T$ to $R \circ U \circ T$ given $U \sqsubseteq S$, for example, we may use $(\backslash x \to {\scriptstyle\blacksquare}\text{-}\mathit{monotonic}\, R\, (U{\sqsubseteq}S\, (T \,{\scriptstyle\blacksquare}\, x)))$ as the reason. It is instructive to study the definition of ${\scriptstyle\blacksquare}$-*monotonic*. After taking R and $s{\subseteq}t$ (a proof of $s \subseteq t$), the function ${\scriptstyle\blacksquare}$-*monotonic* shall return a proof of $R \,{\scriptstyle\blacksquare}\, s \subseteq R \,{\scriptstyle\blacksquare}\, t$. The proof, given a value b and a proof that some a_1 in s is mapped to b through R, shall produce a proof that there exists some value in t that is also mapped to b. The obvious choice of such a value is a_1. Notice how we apply $s{\subseteq}t$ to a_1 and $a_1{\in}s$ to produce a proof that a_1 is also in t.

Another lemma often used without being said is that we can introduce id_R anywhere we need. It can be proved using $\equiv$-*subst*:

$$\begin{aligned}
&\mathit{id\text{-}intro} : \{A\ B : \mathit{Set}\}\{R : B \leftarrow A\} \rightarrow R \sqsupseteq R \circ id_R\\
&\mathit{id\text{-}intro}\ \{\_\}\{\_\}\{R\}\ a\ b\ (\mathit{exists}\ a'\ (a{\equiv}a', bRa')) \ = \\
&\qquad \equiv\text{-}\mathit{subst}\ (\backslash a \rightarrow R\ a\ b)\ (\equiv\text{-}\mathit{sym}\ a{\equiv}a')\ bRa'.
\end{aligned}$$

3.3 Relational Fold

Having defined all the necessary components, we can now define relational fold in terms of functional fold:

$$\begin{aligned}
&\mathit{foldR} : \{A\ B : \mathit{Set}\} \rightarrow (B \leftarrow (A \times B)) \rightarrow \mathsf{P}B \rightarrow (B \leftarrow [A])\\
&\mathit{foldR}\ R\ s \ = \ \mathit{foldr}\ (R \circ_1 (id_R {\times_1} \in))\ s.
\end{aligned}$$

On the top of the list of properties that we wish to have proved is, of course, fold fusion for relational folds:

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

The proof proceeds by converting both sides to functional folds. It is omitted here for brevity but is available online [16]. To use fold fusion, however, there has to be a fold to start with. Luckily, this is hardly a problem, given the following lemma showing that id_R, when instantiated to lists, is a fold:

$$id_R{\sqsupseteq}\mathit{foldR} : \{A : \mathit{Set}\} \rightarrow id_R\ \{[A]\} \ \sqsupseteq \ \mathit{foldR}\ \mathit{cons}\ \mathit{nil},$$

where $\mathit{cons} = \mathit{fun}\ (\mathit{uncurry}\ \_{::}\_)$ and $\mathit{nil} = \mathit{singleton}\ [\,]$. Let us try to construct its proof term. The inclusion $id_R\ \{[A]\} \ \sqsupseteq \ \mathit{foldR}\ \mathit{cons}\ \mathit{nil}$ expands to:

$$\mathit{forall}\ xs\ ys \rightarrow \ \ \mathit{foldR}\ \mathit{cons}\ \mathit{nil}\ xs\ ys \ \ \rightarrow \ \ xs \equiv ys.$$

The proof term of $id_R{\sqsupseteq}\mathit{foldR}$ should be a function which takes xs, ys, and a proof that $\mathit{foldR}\ \mathit{cons}\ \mathit{nil}$ maps xs to ys, and returns a proof of $xs \equiv ys$. When xs is $[\,]$, $\mathit{foldR}\ \mathit{cons}\ \mathit{nil}\ [\,]\ ys$ simplifies to $[\,] \equiv ys$, and we can simply return the proof:

$$id_R{\sqsupseteq}\mathit{foldR}\ [\,]\ \ ys\ \ [\,]{\equiv}ys \ \ = \ \ [\,]{\equiv}ys.$$

Consider the case $a :: xs$. The proposition $\mathit{foldR}\ \mathit{cons}\ \mathit{nil}\ (a\ ::\ xs)\ ys$ expands to $\exists\ (V{\times}[V])\ P$, where $P\ (a', as) = ((a \equiv a'){\times}(\mathit{foldR}\ \mathit{cons}\ \mathit{nil}\ xs\ as)){\times}(\mathit{cons}\ (a, as)\ ys)$.

Given $a :: xs$, ys, and a proof of $\exists\,(V \times [V])\,P$, we should construct a proof that $a :: xs \equiv ys$. We can do so by equational reasoning:

$$
\begin{array}{l}
id_R \sqsupseteq foldR\ (a :: xs)\ ys\ (exists\,(a', as)\,((a{\equiv}a', foldRxsas), a'{::}as{\equiv}ys)) = \\
\quad\quad a :: xs \\
\quad\quad \equiv\langle\quad a{\equiv}a'\ \langle::\rangle\ (id_R \sqsupseteq foldR\ xs\ as\ foldRxsas)\quad\rangle \\
\quad\quad a' :: as \\
\quad\quad \equiv\langle\quad a'{::}as{\equiv}ys\quad\rangle \\
\quad\quad ys \\
\quad\quad \equiv\Box,
\end{array}
$$

where $\langle::\rangle$ is $\equiv$-*cong* applied twice, substituting a for a' and xs for as.

4 Example: Deriving Insertion Sort

We are finally in a position to present our main example: a derivation of insertion sort, adopted from Bird [7].

4.1 Specifying Sort

We first specify what a sorted list is, assuming a datatype *Val* and a binary ordering $\_{\leq}\_ : Val \to Val \to Set$ 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:

$$
\begin{array}{l}
lbound : \mathsf{P}(Val \times [Val]) \\
lbound\,(a, [\,]) \qquad = \top \\
lbound\,(a, b :: xs) = (a \leq b) \times lbound\,(a, xs).
\end{array}
$$

A *coreflexive* is a sub-relation of id_R. The following operator $\_?$ converts a set to a coreflexive, letting the input go through iff it is in the set:

$$
\begin{array}{l}
\_? : \{A : Set\} \to \mathsf{P}A \to (A \leftarrow A) \\
(p\,?)\,a\,b \;=\; (a \equiv b) \times p\,a.
\end{array}
$$

The coreflexive *ordered*?, which lets a list go through iff it is sorted, can then be defined as a fold:

$$
\begin{array}{l}
ordered? \;:\; [Val] \leftarrow [Val] \\
ordered? = foldR\,(cons \circ lbound\,?)\,nil.
\end{array}
$$

We postulate a datatype *Bag*, representing bags of values. Bags are formed by two constructors: $\wr\int : Bag$ and $\_::_{\mathsf{b}}\_ : Val \to Bag \to Bag$. For the derivation to work, we demand that the result of $\_::_{\mathsf{b}}\_$ be distinguishable from the empty bag, and that $\_::_{\mathsf{b}}\_$ be commutative:[8]

$$
\begin{array}{l}
::_{\mathsf{b}}\text{-}nonempty : forall\,\{a\,w\} \to (\wr\int \equiv a ::_{\mathsf{b}} w) \to \bot \\
::_{\mathsf{b}}\text{-}commute\ \ : (a\,b : Val) \to (w : Bag) \to a ::_{\mathsf{b}} (b ::_{\mathsf{b}} w) \equiv b ::_{\mathsf{b}} (a ::_{\mathsf{b}} w).
\end{array}
$$

[8] We can put more constraints on bags, such as that $\_::_{\mathsf{b}}\_$ discards no elements. But the two properties are enough to guarantee that *isort* is included in *ordered*? $\circ$ *permute*.

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

$$\begin{array}{l}
\mathit{bagify}\ :\ [\mathit{Val}] \to \mathit{Bag}\\
\mathit{bagify} = \mathit{foldr}\ \_{::}_{\mathsf{b}}\_\ \wr\!\int.
\end{array}$$

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:

$$\begin{array}{lcl}
\mathit{permute} & : & [\mathit{Val}] \leftarrow [\mathit{Val}]\\
\mathit{permute} & = & (\mathit{fun}\ \mathit{bagify})^{\smile} \circ \mathit{fun}\ \mathit{bagify},\\
\\
\mathit{sort} & : & [\mathit{Val}] \leftarrow [\mathit{Val}]\\
\mathit{sort} & = & \mathit{ordered?} \circ \mathit{permute}.
\end{array}$$

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

4.2 The Derivation

The derivation begins with observing that *permute* can be turned into a fold. We first introduce an id_R by *id-intro*, followed by the lemma $\mathit{id}_R{\sqsupseteq}\mathit{foldR}$, and fold fusion:

$$\begin{array}{l}
\mathit{perm\text{-}der} : \exists_1\,([\mathit{Val}] \leftarrow [\mathit{Val}])\,(\backslash \mathit{perm} \to \mathit{permute} \sqsupseteq \mathit{perm})\\
\mathit{perm\text{-}der} = \mathit{exists}_1\ \_\ (\quad \begin{array}[t]{ll}
 & \mathit{permute}\\
\sqsupseteq\langle & \mathit{id\text{-}intro} \quad \rangle\\
 & \mathit{permute} \circ \mathit{id}_R\\
\sqsupseteq\langle & (\backslash \mathit{xs} \to {\bullet}\text{-}\mathit{monotonic}\ \mathit{permute}\ (\mathit{id}_R{\sqsupseteq}\mathit{foldR}\ \mathit{xs})) \quad \rangle\\
 & \mathit{permute} \circ \mathit{foldR}\ \mathit{cons}\ \mathit{nil}\\
\sqsupseteq\langle & \mathit{foldR\text{-}fusion}\ \mathit{permute}\ \mathit{perm\text{-}step}\ \mathit{perm\text{-}base} \quad \rangle\\
 & \mathit{foldR}\ \mathit{combine}\ \mathit{nil}\\
\sqsupseteq\Box), &
\end{array}
\end{array}$$

where $\exists_1$ is a *Set*1 variant of $\exists$, with extraction functions $\mathit{witness}_1$ and proof_1. The relation *combine* can be defined as follows:

$$\begin{array}{l}
\mathit{combine} : [\mathit{Val}] \leftarrow (\mathit{Val} \times [\mathit{Val}])\\
\mathit{combine}\,(a, \mathit{xs}) \qquad = \mathit{cons}\,(a, \mathit{xs}) \cup \mathit{combine}'\,(a, \mathit{xs}),\\
\mathit{combine}' : [\mathit{Val}] \leftarrow (\mathit{Val} \times [\mathit{Val}])\\
\mathit{combine}'\,(a, [\,]) \qquad = \backslash \mathit{ys} \to \bot\\
\mathit{combine}'\,(a, b :: \mathit{xs}) = (\backslash \mathit{zs} \to \mathit{cons}\,(b, \mathit{zs})) \bullet \mathit{combine}\,(a, \mathit{xs}).
\end{array}$$

Given (a, xs), it inserts a into an arbitrary position of xs. For the *foldR-fusion* to work, we have to provide two proofs:

$$\begin{array}{l}
\mathit{perm\text{-}step} : \mathit{permute} \circ \mathit{cons} \sqsupseteq \mathit{combine} \circ (\mathit{id}_R \times \mathit{permute})\\
\mathit{perm\text{-}base} : \mathit{permute} \bullet \mathit{nil} \supseteq \mathit{nil}.
\end{array}$$

But the real work is done in proving that shuffling the input list does not change the result of *bagify*:

$$\begin{aligned}&\mathit{bagify\text{-}homo} : (a : \mathit{Val}) \to (\mathit{xs}\ \mathit{ys} : [\mathit{Val}]) \to \\ &\qquad \mathit{combine}\,(a, \mathit{xs})\ \mathit{ys} \to \mathit{bagify}\,(a :: \mathit{xs}) \equiv \mathit{bagify}\ \mathit{ys}.\end{aligned}$$

It is when proving this lemma that we need $::_{\mathsf{b}}$ *-commute*.

After the reasoning above, we have at our hands:

$$\begin{aligned}&\mathit{perm}\ :\ [\mathit{Val}] \leftarrow [\mathit{Val}]\\ &\mathit{perm} = \mathit{witness}_1\ \mathit{perm\text{-}der},\end{aligned}$$

$$\begin{aligned}&\mathit{permute\text{-}is\text{-}fold}\ :\ \mathit{permute} \sqsupseteq \mathit{perm}\\ &\mathit{permute\text{-}is\text{-}fold} = \mathit{proof}_1\ \mathit{perm\text{-}der}.\end{aligned}$$

Therefore, *perm* = *foldR combine nil*, while *permute-is-fold* is a proof that *perm* refines *permute*.

Now that *permute* can be refined to a fold, a natural step to try is to fuse *ordered*? into the fold. We derive:

$$\begin{array}{l}\mathit{sort\text{-}der} : \exists\,([\mathit{Val}] \to [\mathit{Val}])\,(\backslash f \to \mathit{ordered?} \circ \mathit{permute} \sqsupseteq \mathit{fun}\,f)\\ \mathit{sort\text{-}der} = \mathit{exists}\ \_\\ \qquad\qquad\begin{array}{ll}(& \mathit{ordered?} \circ \mathit{permute}\\ \sqsupseteq\langle & (\backslash \mathit{xs} \to \centerdot\text{-}\mathit{monotonic}\ \mathit{ordered?}\,(\mathit{permute\text{-}is\text{-}fold}\ \mathit{xs}))\quad\rangle\\ & \mathit{ordered?} \circ \mathit{perm}\\ \sqsupseteq\langle & \sqsupseteq\text{-}\mathit{refl}\quad\rangle\\ & \mathit{ordered?} \circ \mathit{foldR}\ \mathit{combine}\ \mathit{nil}\\ \sqsupseteq\langle & \mathit{foldR\text{-}fusion}\ \mathit{ordered?}\ \mathit{ins\text{-}step}\ \mathit{ins\text{-}base}\quad\rangle\\ & \mathit{foldR}\,(\mathit{fun}\,(\mathit{uncurry}\ \mathit{insert}))\ \mathit{nil}\\ \sqsupseteq\langle & \mathit{foldR\text{-}to\text{-}foldr}\ \mathit{insert}\ [\,]\quad\rangle\\ & \mathit{fun}\,(\mathit{foldr}\ \mathit{insert}\ [\,])\\ \sqsupseteq\Box). & \end{array}\end{array}$$

The function *insert* follows the usual definition:

$$\begin{array}{l}\mathit{insert} : \mathit{Val} \to [\mathit{Val}] \to [\mathit{Val}]\\ \mathit{insert}\ a\ [\,] = a :: [\,]\\ \mathit{insert}\ a\ (b :: \mathit{xs})\ \mathbf{with}\ a \leq ?b\\ \ldots \qquad \mid \mathit{yes}\ a{\leq}b\ \ =\ \ a :: b :: \mathit{xs}\\ \ldots \qquad \mid \mathit{no}\ \ a{\not\leq}b\ \ =\ \ b :: \mathit{insert}\ a\ \mathit{xs},\end{array}$$

where $a \leq ?b$ determines whether $a \leq b$, whose result is case-matched by the **with** notation. The fusion conditions are:

$$\begin{array}{l}\mathit{ins\text{-}step} : \mathit{ordered?} \circ \mathit{combine} \sqsupseteq \mathit{fun}\,(\mathit{uncurry}\ \mathit{insert}) \circ (\mathit{id}_R \times \mathit{ordered?})\\ \mathit{ins\text{-}base} : \mathit{ordered?} \centerdot \mathit{nil} \supseteq \mathit{nil}.\end{array}$$

Finally, *foldR-to-foldr* is a small lemma allowing us to convert a relational fold to a functional fold, provided that its arguments have been refined to a function and a singleton set already:

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

We have thus derived $isort = witness\ sort\text{-}der = foldr\ insert\ [\,]$, while at the same time proved that it meets the specification $ordered? \circ permute$. The details of the proofs are available online [16]. The library code defining sets, relations, folds, and their properties, amounts to about 800 lines. The main derivation of *isort* is not long. Proving the fusion condition *ins-step* and its related properties turned out to take some hard work and eventually adds up to about 700 lines of code. The interactive mode was of great help — the proof would have been difficult to construct by hand.

5 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. It also demonstrates that dependent types are expressive enough to demand that a program satisfies an input/output relation. An interesting way to construct the corresponding proof term, which would be difficult to build otherwise, is derivation.

McKinna and Burstall's paper on "Deliverables" [15] 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 early paper [18] to the impressive four-colour theorem development (including the development of a verified compiler). Our contribution is more modest — we aim at formally checked but still readable Algebra-of-Programming style derivations.

The concept of Inductive Families [12], 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 [13, Ch. 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 [2], Coq [10], Dependent ML [23], Agda [17], Ωmega [19], Epigram [14], and the GADT extension [9] to Haskell. It is believed that dependent types have an important role in the next generation of programming languages [20].

Acknowledgements We are grateful to Nils Anders Danielsson for pointing out typos and giving valuable suggestions regarding the presentation.

References

1. T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. Draft, 2005.
2. L. Augustsson. Cayenne – a language with dependent types. In *ICFP'98*, pages 239–250, 1998.

3. L. Augustsson. Equality proofs in Cayenne. Chalmers Univ. of Tech., 1999.
4. R. C. Backhouse et al. Relational catamorphisms. In *IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier, 1991.
5. R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller et al., editors, *Formal Program Development.*, number 755 in LNCS, pages 7–42. Springer-Verlag, January 1992.
6. R. S. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, April 1989.
7. R. S. Bird. Functional algorithm design. *Science of Computer Programming*, 26:15–31, 1996.
8. R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.
9. J. Cheney and R. Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, 2003.
10. The Coq Development Team, LogiCal Project. *The Coq Proof Assistant Reference Manual*, 2006.
11. O. de Moor and G. Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269(1-2):135–162, 2001.
12. P. Dybjer. Inductive families. *Formal Aspects of Computing*, pages 440–465, 1994.
13. C. Gonzalía. *Relations in Dependent Type Theory*. PhD thesis, Chalmers Univ. of Tech., 2006.
14. C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
15. J. McKinna and R. M. Burstall. Deliverables: A categorial approach to program development in type theory. In *MFCS'93: Int. Symp. on Mathematical Foundations of Computer Science*, pages 32–67. Springer-Verlag, 1993.
16. S.-C. Mu, H.-S. Ko, and P. Jansson. AoPA: Algebra of programming in Agda. `http://www.iis.sinica.edu.tw/~scm/2008/aopa/`.
17. U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers Univ. of Tech., 2007.
18. C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *POPL'89*, Austin, Jan. 1989. ACM.
19. T. Sheard. Programming in *Ω*mega. The 2nd Central European Functional Programming School, June 2007.
20. T. Sweeney. The next mainstream programming language: a game developer's perspective. Invited Talk, POPL'06, January 2006.
21. The Agda Team. The Agda Wiki. Available at `http://www.cs.chalmers.se/~ulfn/Agda/`, 2007.
22. R. Verhoeven and R. C. Backhouse. Towards tool support for program verification and construction. In *World Congress on Formal Methods*, pages 1128–1146, 1999.
23. H. Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, March 2007.
24. T. Yokoyama, Z. Hu, and M. Takeichi. Yicho - a system for programming program calculations. In *The 3rd Asian Workshop on Programming Languages and Systems (APLAS 2002)*, pages 366–382, 2002.