Declarative Pearl: Deriving Monadic Quicksort

Shin-Cheng Mu[1] and Tsung-Ju Chiang[2]

[1] Academia Sinica, Taiwan
[2] National Taiwan University, Taiwan

Abstract. To demonstrate derivation of monadic programs, we present a specification of sorting using the non-determinism monad, and derive pure quicksort on lists and state-monadic quicksort on arrays. In the derivation one may switch between point-free and pointwise styles, and deploy techniques familiar to functional programmers such as pattern matching and induction on structures or on sizes. Derivation of stateful programs resembles reasoning backwards from the postcondition.

Keywords: monads · program derivation · equational reasoning · non-determinism · state · quicksort

1 Introduction

This pearl presents two derivations of quicksort. The purpose is to demonstrate reasoning and derivation of monadic programs. In the first derivation we present a specification of sorting using the non-determinism monad, from which we derive a pure function that sorts a list. In the second derivation we derive an imperative algorithm, expressed in terms of the state monad, that sorts an array.

Before we dive into the derivations, we shall explain our motivation. Program derivation is the technique of formally constructing a program from a problem specification. In functional derivation, the specification is a function that obviously matches the problem description, albeit inefficiently. It is then stepwise transformed to a program that is efficient enough, where every step is justified by mathematical properties guaranteeing that the program *equals* the specification, that is, for all inputs they compute exactly the same output.

It often happens, for certain problem, that several answers are equally preferred. In sorting, for example, the array to be sorted might contain items with identical keys. It would be inflexible, if not impossible, to decide in the specification how to resolve the tie: it is hard to predict how quicksort arranges items with identical keys before actually deriving quicksort.[3] Such problems are better modelled as non-deterministic mappings from the input to all valid outputs. The derived program no longer equals but *refines* the specification.[4]

[3] Unless we confine ourselves to stable sorting.
[4] This is standard in imperative program derivation — Dijkstra [6] argued that we should take non-determinism as default and determinism as a special case.

To cope with non-determinism, there was a trend in the 90's generalising from functions to relations [1,4]. Although these relational calculi are, for advocates including the authors of this paper, concise and elegant, for those who were not following this line of development, these calculi are hard to comprehend and use. People, in their first and often only exposure to the calculi, often complained that the notations are too bizarre, and reasoning with inequality (refinement) too complex. One source of difficulties is that notations of relational calculus are usually *point-free* — that is, about composing relations instead of applying relations to arguments. There have been attempts (e.g [13,5]) designing *pointwise* notations, which functional programmers are more familiar with. Proposals along this line tend to exhibit confusion when functions are applied to non-deterministic values — β-reduction and η-conversion do not hold. One example [13] is that $(\lambda x \to x - x)\ (0 \,[\!]\, 1)$, where $([\!])$ denotes non-deterministic choice, always yields 0, while $(0 \,[\!]\, 1) - (0 \,[\!]\, 1)$ could be 0, 1, or -1.

Preceding the development of relations for program derivation, another way to model non-determinism has gained popularity. Monads [12] were introduced into functional programming as a way to rigorously talk about side effects including IO, state, exception, and non-determinism. Although they are considered one of the main obstacles in learning functional programming (in particular Haskell), monads have gained wide acceptance. In this pearl we propose a calculus of program derivation based on monads — essentially moving to a Kleisli category. Problem specifications are given as Kleisli arrows for non-deterministic monads, to be refined to deterministic functional programs through calculation. One of the benefits is that functional programmers may deploy techniques they are familiar with when reasoning about and deriving programs. These include both point-free and pointwise reasoning, and induction on structures or sizes of data types. An additional benefit of using monads is that we may talk about effects other than non-determinism. We demonstrate how to, from a specification of quicksort on lists, construct the imperative quicksort for arrays. All the derivations and theorems in this pearl are verified in the dependently typed programming language Agda.[5]

2 Monads

A monad consists of a type constructor $m :: * \to *$ paired with two operators, can be modelled in Haskell as a type class:

VMonad vlift vbind

$$\begin{aligned}&\textbf{class}\ \mathsf{Monad}\ m\ \textbf{where}\\ &\quad \{\cdot\} \quad :: a \to m\ a\\ &\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b \quad .\end{aligned}$$

The operator $\{\cdot\}$ is usually called *return* or *unit*. Since it is used pervasively in this pearl, we use a shorter notation for brevity. One can either think of it

[5] https://scm.iis.sinica.edu.tw/home/2020/deriving-monadic-quicksort/

as mimicking the notation for a singleton set, or C-style syntax for a block of effectful program. They should satisfy the following *monad laws*:

$$\begin{aligned} m \gg\!= \{\cdot\} &= m \quad , \\ \{x\} \gg\!= f &= f\ x \quad , \\ (m \gg\!= f) \gg\!= g &= m \gg\!= (\lambda x \to f\ x \gg\!= g) \quad . \end{aligned}$$

A standard operator $(\gg) :: \mathsf{Monad}\ m \Rightarrow m\ a \to m\ b \to m\ b$, defined by $m_1 \gg m_2 = m_1 \gg\!= \lambda\_ \to m_2$, is handy when we do not need the result of m_1. Monadic functions can be combined by Kleisli composition $(\gg\!\!>)$, defined by $f \gg\!\!> g = \lambda x \to f\ x \gg\!= g$.

Monads usually come with additional operators corresponding to the effects they provide. Regarding non-determinism, we assume two operators $\emptyset$ and $([])$, respectively denoting failure and non-deterministic choice:

VMonadPlus
vmpepsilon
vmpadd

$$\begin{aligned} &\textbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadPlus}\ m\ \textbf{where} \\ &\quad \emptyset \quad :: m\ a \\ &\quad (]\!]) :: m\ a \to m\ a \to m\ a \quad . \end{aligned}$$

It might be a good time to note that this pearl uses type classes for two purposes: firstly, to be explicit about the effects a program uses. Secondly, the notation implies that it does not matter which actual implementation we use for m, as long as it satisfies all the properties we demand — as Gibbons and Hinze [7] proposed, we use the properties, not the implementations, when reasoning about programs. The style of reasoning in this pearl is not tied to type classes or Haskell, and we do not strictly follow the particularities of type classes in the current Haskell standard.[6]

It is usually assumed that $(]\!])$ is associative with $\emptyset$ as its identity:

$$\emptyset\ ]\!]\ m = m\ =\ m\ ]\!]\ \emptyset \quad , \qquad (m_1\ ]\!]\ m_2)\ ]\!]\ m_3 = m_1\ ]\!]\ (m_2\ ]\!]\ m_3) \quad .$$

For the purpose of this pearl, we also demand that $(]\!])$ be idempotent and commutative. That is, $m\ ]\!]\ m = m$ and $m\ ]\!]\ n = n\ ]\!]\ m$. Efficient implementations of such monads have been proposed (e.g. [10]). However, we use non-determinism monad only in specification. The derived programs are always deterministic.

The laws below concern interaction between non-determinism and $(\gg\!=)$:

$$\emptyset \gg\!= f = \emptyset \quad , \tag{1}$$

$$m \gg \emptyset = \emptyset \quad , \tag{2}$$

$$(m_1\ ]\!]\ m_2) \gg\!= f = (m_1 \gg\!= f)\ ]\!]\ (m_2 \gg\!= f) \quad , \tag{3}$$

$$m \gg\!= (\lambda x \to f_1\ x\ ]\!]\ f_2\ x) = (m \gg\!= f_1)\ ]\!]\ (m \gg\!= f_2) \quad . \tag{4}$$

Left-zero (1) and left-distributivity (3) are standard — the latter says that $(]\!])$ is algebraic. When mixed with state, right-zero (2) and right-distributivity (4) imply that each non-deterministic branch has its own copy of the state [14].

[6] For example, we overlook that a Monad must also be Applicative, MonadPlus be Alternative, and that functional dependency is needed in a number of places.

3 Specification

We are now ready to present a monadic specification of sorting. Bird [3] demonstrated how to derive various sorting algorithms from relational specifications. In Section 4 and 5 we show how quicksort can be derived in our monadic calculus.

We assume a type Elm (for "elements") associated with a total preorder ($\leqslant$). To sort a list $xs :: \mathsf{List\ Elm}$ is to choose, among all permutation of xs, those that are sorted:

slowsort

$$\begin{aligned} &\mathit{slowsort} :: \mathsf{MonadPlus}\ m \Rightarrow \mathsf{List\ Elm} \to m\ (\mathsf{List\ Elm}) \\ &\mathit{slowsort} = \mathit{perm} \gg\!\!> \mathit{filt}\ \mathit{sorted}\ \ , \end{aligned}$$

where $\mathit{perm} :: \mathsf{MonadPlus}\ m \Rightarrow \mathsf{List}\ a \to m\ (\mathsf{List}\ a)$ non-deterministically computes a permutation of its input, $\mathit{sorted} :: \mathsf{List\ Elm} \to \mathsf{Bool}$ checks whether a list is sorted, and $\mathit{filt}\ p\ x$ returns x if $p\ x$ holds, and fails otherwise:

guardBy

$$\begin{aligned} &\mathit{filt} :: \mathsf{MonadPlus}\ m \Rightarrow (a \to \mathsf{Bool}) \to a \to m\ a \\ &\mathit{filt}\ p\ x = \mathit{guard}\ (p\ x) \gg \{x\}\ \ . \end{aligned}$$

The function $\mathit{guard}\ b = \mathbf{if}\ b\ \mathbf{then}\ \{\}\ \mathbf{else}\ \emptyset$ is standard. The predicate $\mathit{sorted} :: \mathsf{List\ Elm} \to \mathsf{Bool}$ can be defined by:

$$\begin{aligned} \mathit{sorted}\ [\,] &= \mathsf{True} \\ \mathit{sorted}\ (x : xs) &= \mathit{all}\ (x \leqslant)\ xs \wedge \mathit{sorted}\ xs\ \ . \end{aligned}$$

The following property can be proved by a routine induction on ys:

$$\begin{aligned} &\mathit{sorted}\ (ys +\!\!+ [x] +\!\!+ zs)\ \equiv \\ &\quad \mathit{sorted}\ ys \wedge \mathit{sorted}\ zs \wedge \mathit{all}\ (\leqslant x)\ ys \wedge \mathit{all}\ (x \leqslant)\ zs\ \ . \end{aligned} \tag{5}$$

Now we consider the permutation phase. As shown by Bird [3], what sorting algorithm we end up deriving is often driven by how the permutation phase is performed. The following definition of perm, for example:

$$\begin{aligned} \mathit{perm}\ [\,] &= \{[\,]\} \\ \mathit{perm}\ (x : xs) &= \mathit{perm}\ xs \gg\!\!= \mathit{insert}\ x\ \ , \end{aligned}$$

where $\mathit{insert}\ x\ xs$ non-deterministically inserts x into xs, would lead us to insertion sort. To derive quicksort, we use an alternative definition of perm:

$$\begin{aligned} &\mathit{perm} :: \mathsf{MonadPlus}\ m \Rightarrow \mathsf{List}\ a \to m\ (\mathsf{List}\ a) \\ &\mathit{perm}\ [\,] = \{[\,]\} \\ &\mathit{perm}\ (x : xs) = \mathit{split}\ xs \gg\!\!= \lambda(ys, zs) \to \mathit{liftM2}\ (+\!\!+[x]+\!\!+)\ (\mathit{perm}\ ys)\ (\mathit{perm}\ zs)\ \ . \end{aligned}$$

where $\mathit{liftM2}\ (\oplus)\ m_1\ m_2 = m_1 \gg\!\!= \lambda x_1 \to m_2 \gg\!\!= \lambda x_2 \to \{x_1 \oplus x_2\}$, and split non-deterministically splits a list. When the input has more than one element, we split the tail into two, permute them separately, and insert the head in the middle. The monadic function split is given by:

split

$$\begin{aligned} &\mathit{split} :: \mathsf{MonadPlus}\ m \Rightarrow \mathsf{List}\ a \to m\ (\mathsf{List}\ a \times \mathsf{List}\ a) \\ &\mathit{split}\ [\,] \quad\ = \{([\,],[\,])\} \\ &\mathit{split}\ (x : xs) = \mathit{split}\ xs \gg\!= \lambda(ys, zs) \to \{(x : ys, zs)\} \,[\!]\, \{(ys, x : zs)\} \quad . \end{aligned}$$

This completes the specification. One may argue that the second definition of *perm* is not one that, as stated in Section 1, "obviously" implied by the problem description. Bird [3] derived the second one from the first in a relational setting, and we can also show that the two definitions are equivalent.

4 Quicksort on Lists

In this section we derive a divide-and-conquer property of *slowsort*. It allows us to refine *slowsort* to the well-known recursive definition of quicksort on lists, and is also used in the next section to construct quicksort on arrays.

Refinement We will need to first define our concept of program refinement. We abuse notations from set theory and define:

RefinesPlus

$$m_1 \subseteq m_2 \ \equiv\ m_1 \,[\!]\, m_2 = m_2 \quad .$$

The righthand side $m_1 \,[\!]\, m_2 = m_2$ says that every result of m_1 is a possible result of m_2. When $m_1 \subseteq m_2$, we say that m_1 *refines* m_1, m_2 can be *refined to* m_1, or that m_2 *subsumes* m_1. Note that this definition applies not only to the non-determinism monad, but to monads having other effects as well. We denote $(\subseteq)$ lifted to functions by $(\dot{\subseteq})$:

RefinesPlusF

$$f \mathbin{\dot{\subseteq}} g \ =\ (\forall x : f\ x \subseteq g\ x) \quad .$$

That is, f refines g if $f\ x$ refines $g\ x$ for all x.When we use this notation, f and g are always functions returning monads, which is sufficient for this pearl.

One can show that the definition of $(\subseteq)$ is equivalent to $m_1 \subseteq m_2 \equiv (\exists\, n : m_1 \,[\!]\, n = m_2)$, and that $(\subseteq)$ and $(\dot{\subseteq})$ are both reflexive, transitive, and anti-symmetric $(m \subseteq n \wedge n \subseteq m \equiv n = m)$. Furthermore, $(\gg\!=)$ respects refinement:

Lemma 1. *Bind* $(\gg\!=)$ *is monotonic with respect to* $(\subseteq)$*. That is,* $m_1 \subseteq m_2 \Rightarrow m_1 \gg\!= f \subseteq m_2 \gg\!= f$*, and* $f_1 \mathbin{\dot{\subseteq}} f_2 \Rightarrow m \gg\!= f_1 \subseteq m \gg\!= f_2$*.*

Having Lemma 1 allows us to refine programs in a compositional manner. The proof of Lemma 1 makes use of (3) and (4).

Commutativity and *guard* We say that m and n commute if

$$m \gg\!= \lambda x \to n \gg\!= \lambda y \to f\ x\ y \ =\ n \gg\!= \lambda y \to m \gg\!= \lambda x \to f\ x\ y \quad .$$

It can be proved that *guard* p commutes with all m if non-determinism is the only effect in m — a property we will need many times. Furthermore, having

right-zero (2) and right-distributivity (4), in addition to other laws, one can prove that non-determinism commutes with other effects. In particular, non-determinism commutes with state.

We mention two more properties about *guard*: $guard\ (p \wedge q)$ can be split into two, and *guard*s with complementary predicates can be refined to **if**:

guard_disjoint

$$guard\ (p \wedge q) \;=\; guard\ p \gg guard\ q \quad , \tag{6}$$

$$(guard\ p \gg m_1) \,[\!]\, (guard\ (\neg \cdot p) \gg m_2) \;\supseteq\; \textbf{if}\ p\ \textbf{then}\ m_1\ \textbf{else}\ m_2 \quad . \tag{7}$$

Divide-and-Conquer Back to *slowsort*. We proceed with usual routine in functional programming: case-analysis on the input. For the base case, $slowsort\ [\,] = \{[\,]\}$. For the inductive case, the crucial step is the commutativity of *guard*:

$$\begin{array}{ll}
& slowsort\ (p : xs) \\
= & \quad \{\ \text{expanding definitions, monad laws}\ \} \\
& split\ xs \gg\!= \lambda(ys, zs) \rightarrow \\
& perm\ ys \gg\!= \lambda ys' \rightarrow perm\ zs \gg\!= \lambda zs' \rightarrow \\
& filt\ sorted\ (ys' +\!\!+ [p] +\!\!+ zs') \\
= & \quad \{\ \text{by (5)}\ \} \\
& split\ xs \gg\!= \lambda(ys, zs) \rightarrow \\
& perm\ ys \gg\!= \lambda ys' \rightarrow perm\ zs \gg\!= \lambda zs' \rightarrow \\
& guard\ (sorted\ ys' \wedge sorted\ zs' \wedge all\ (\leqslant p)\ ys' \wedge all\ (p \leqslant)\ zs') \gg \\
& \{ys' +\!\!+ [p] +\!\!+ zs'\} \\
= & \quad \{\ \text{(6) and that } guard \text{ commutes with non-determinism}\ \} \\
& split\ xs \gg\!= \lambda(ys, zs) \rightarrow guard\ (all\ (\leqslant p)\ ys \wedge all\ (p \leqslant)\ zs') \gg \\
& (perm\ ys \gg\!= filt\ sorted) \gg\!= \lambda ys' \rightarrow \\
& (perm\ zs \gg\!= filt\ sorted) \gg\!= \lambda zs' \rightarrow \\
& \{ys' +\!\!+ [p] +\!\!+ zs'\} \quad .
\end{array}$$

Provided that we can construct a function *partition* such that

divide_and_conquer_lemma1

divide_and_conquer_lemma2

$$\{partition\ p\ xs\} \;\subseteq\; split\ xs \gg\!= filt\ (\lambda(ys, zs) \rightarrow all\ (\leqslant p)\ ys \wedge all\ (p \leqslant)\ zs) \quad ,$$

we have established the following divide-and-conquer property:

divide_and_conquer

$$\begin{array}{lll}
slowsort\ (p : xs) & \supseteq & \{partition\ p\ xs\} \gg\!= \lambda(ys, zs) \rightarrow \\
& & slowsort\ ys \gg\!= \lambda ys' \rightarrow slowsort\ zs \gg\!= \lambda zs' \rightarrow \\
& & \{ys' +\!\!+ [p] +\!\!+ zs'\} \quad .
\end{array} \tag{8}$$

The derivation of *partition* proceeds by induction on the input. In the case for $xs := x : xs$ we need to refine two guarded choices, $(guard\ (x \leqslant p) \gg \{x : ys, zs\}) \,[\!]\, (guard\ (p \leqslant x) \gg \{ys, x : zs\})$, to an **if** branching. When x and p equal, the specification allows us to place x in either partition. For no particular reason, we choose the left partition. That gives us:

$$\begin{array}{lll}
partition\ p\ [\,] & = ([\,], [\,]) & \\
partition\ p\ (x : xs) & = \textbf{let}\ (ys, zs) = partition\ p\ xs & \\
& \quad \textbf{in if}\ x \leqslant p\ \textbf{then}\ (x : ys, zs)\ \textbf{else}\ (ys, x : zs) \quad .
\end{array}$$

Having *partition* derived, it takes only a routine induction on the length of input lists to show that $\{\cdot\} \cdot qsort \mathrel{\dot\subseteq} slowsort$, where *qsort* is given by:

$$\begin{aligned} qsort\ [\,] &= [\,] \\ qsort\ (p : xs) &= \mathbf{let}\ (ys, zs) = partition\ p\ xs \\ &\quad\ \mathbf{in}\ qsort\ ys +\!\!+ [p] +\!\!+ qsort\ zs \quad . \end{aligned}$$

As is typical in program derivation, the termination of derived program is shown separately afterwards. In this case, *qsort* terminates because the input list decreases in size in every recursive call — for that we need to show that, in the call to *partition*, the sum of lengths of *ys* and *zs* equals that of *xs*.

5 Quicksort on Arrays

One of the advantages of using a monadic calculus is that we can integrate effects other than non-determinism into the program we derive. In this section we derive an imperative quicksort on arrays, based on previously established properties.

5.1 Operations on Arrays

We assume that our state is an Int-indexed, unbounded array containing elements of type e, with two operations that, given an index, respectively read from and write to the array:

$$\begin{aligned} &\mathbf{class}\ \mathsf{Monad}\ m \Rightarrow \mathsf{MonadArr}\ e\ m\ \mathbf{where} \\ &\quad read\ \ :: \mathsf{Int} \to m\ e \\ &\quad write :: \mathsf{Int} \to e \to m\ ()\ \ . \end{aligned}$$

They are assumed to satisfy the following laws:

$$\begin{aligned} &\textbf{read-write:} && read\ i \gg\!= write\ i\ &&=\ \{()\}\ , \\ &\textbf{write-read:} && write\ i\ x \gg read\ i\ &&=\ write\ i\ x \gg \{x\}\ , \\ &\textbf{write-write:} && write\ i\ x \gg write\ i\ x'\ &&=\ write\ i\ x'\ , \\ &\textbf{read-read:} && read\ i \gg\!= \lambda x \to read\ i \gg\!= \lambda x' \to f\ x\ x'\ &&= \\ & && \qquad read\ i \gg\!= \lambda x \to f\ x\ x\quad . \end{aligned}$$

Furthermore, we assume that (1) *read i* and *read j* commute; (2) *write i x* and *write j y* commute if $i \neq j$; (3) *write i x* and *read j* commute if $i \neq j$.

More operations defined in terms of *read* and *write* are shown in Figure 1, where *#xs* abbreviates *length xs*. The function *readList i n*, where *n* is a natural number, returns a list containing the *n* elements in the array starting from index *i*. Conversely, *writeList i xs* writes the list *xs* to the array with the first element being at index *i*. In imperative programming we often store sequences of data into an array and return the length of the data. Thus, functions *writeL*, *write2L* and *write3L* store lists into the array before returning their lengths. These *read*

$$
\begin{aligned}
&\mathit{readList} :: \mathsf{MonadArr}\ e\ m \Rightarrow \mathsf{Int} \to \mathsf{Nat} \to m\ (\mathsf{List}\ e)\\
&\mathit{readList}\ i\ 0 \qquad\ = \{[]\}\\
&\mathit{readList}\ i\ (1+k) = \mathit{liftM2}\ (:)\ (\mathit{read}\ i)\ (\mathit{readList}\ (i+1)\ k)\ \ ,\\
&\mathit{writeList} :: \mathsf{MonadArr}\ e\ m \Rightarrow \mathsf{Int} \to \mathsf{List}\ e \to m\ ()\\
&\mathit{writeList}\ i\ [] \qquad\ = \{()\}\\
&\mathit{writeList}\ i\ (x:xs) = \mathit{write}\ i\ x \gg \mathit{writeList}\ (i+1)\ xs\ \ ,\\
&\mathit{writeL}\ \ i\ xs \qquad\ = \mathit{writeList}\ i\ xs \gg \{\#xs\}\ \ ,\\
&\mathit{write2L}\ i\ (xs, ys) \quad = \mathit{writeList}\ i\ (xs +\!\!+ ys) \gg \{(\#xs, \#ys)\}\ \ ,\\
&\mathit{write3L}\ i\ (xs, ys, zs) = \mathit{writeList}\ i\ (xs +\!\!+ ys +\!\!+ zs) \gg \{(\#xs, \#ys, \#zs)\}\ \ .\\
&\mathit{swap}\ i\ j = \mathit{read}\ i \gg\!= \lambda x \to \mathit{read}\ j \gg\!= \lambda y \to \mathit{write}\ i\ y \gg \mathit{write}\ j\ x\ \ .
\end{aligned}
$$

Fig. 1: Operations for reading and writing chunks of data.

and *write* family of functions are used only in the specification; the algorithm we construct should only mutate the array by *swap*ing elements.

Among the many properties of *readList* and *writeList* that can be induced from their definitions, the following will be used in a number of crucial steps:

$$\mathit{writeList}\ i\ (xs +\!\!+ ys)\ =\ \mathit{writeList}\ i\ xs \gg \mathit{writeList}\ (i + \#xs)\ ys\ \ . \tag{9}$$

A function $f :: \mathsf{List}\ a \to m\ (\mathsf{List}\ a)$ is said to be *length preserving* if $f\ xs \gg\!= \lambda ys \to \{(ys, \#ys)\} = f\ xs \gg\!= \lambda ys \to \{(ys, \#xs)\}$. It can be proved that *perm*, and thus *slowsort*, are length preserving.

On "composing monads" In the sections to follow, some readers may have concern seeing *perm*, having class constraint $\mathsf{MonadPlus}\ m$, and some other code having constraint $\mathsf{MonadArr}\ e\ m$ in the same expression. This is totally fine: mixing two such subterms simply results in an expression having constraint $(\mathsf{MonadPlus}\ m, \mathsf{MonadArr}\ e\ m)$. No *lift*ing is necessary.

We use type classes to make it clear that we do not specify what exact monad *perm* is implemented with. It could be one monolithic monad, a monad built from monad transformers [8], or a free monad interpreted by effect handlers [11]. All theorems and derivations about *perm* hold regardless of the actual monad, as long as the monad satisfies all properties we demand.

5.2 Partitioning an Array

While the list-based *partition* is relatively intuitive, partitioning an array *in-place* (that is, using at most $O(1)$ additional space) is known to be a tricky phase of array-based quicksort. Therefore we commence our discussion from deriving in-place array partitioning from the list version. The partition algorithm we end up deriving is known as the *Lomuto scheme* [2], as opposed to Hoare's [9].

Specification There are two issues to deal with before we present a specification for an imperative, array-based partitioning, based on list-based *partition*.

Firstly, *partition* is not tail-recursive, while many linear-time array algorithms are implemented as a tail-recursive for-loop. Thus we apply the standard trick constructing a tail-recursive algorithm by introducing accumulating parameters. Define (we write the input/outputs of *partition* in bold font for clarity):

$$\begin{aligned}&\mathit{partl} :: \mathsf{Elm} \to (\mathsf{List\ Elm} \times \mathsf{List\ Elm} \times \mathsf{List\ Elm}) \to (\mathsf{List\ Elm} \times \mathsf{List\ Elm})\\ &\mathit{partl}\ p\ (ys, zs, \boldsymbol{xs}) = \mathbf{let}\ (\boldsymbol{us}, \boldsymbol{vs}) = \mathit{partition}\ p\ \boldsymbol{xs}\\ &\qquad\qquad\qquad\qquad\quad \mathbf{in}\ (ys +\!\!+ \boldsymbol{us}, zs +\!\!+ \boldsymbol{vs})\quad .\end{aligned}$$

In words, $\mathit{partl}\ p\ (ys, zs, \boldsymbol{xs})$ partitions $\boldsymbol{xs}$ into $(\boldsymbol{us}, \boldsymbol{vs})$ with respect to pivot p, but appends ys and zs respectively to $\boldsymbol{us}$ and $\boldsymbol{vs}$. It is a generalisation of *partition* because $\mathit{partition}\ p\ xs = \mathit{partl}\ p\ ([\,], [\,], xs)$. By routine calculation exploiting associativity of $(+\!\!+)$, we can derive a tail-recursive definition of *partl*:

$$\begin{aligned}&\mathit{partl}\ p\ (ys, zs, [\,]) \qquad = (ys, zs)\\ &\mathit{partl}\ p\ (ys, zs, \boldsymbol{x} : \boldsymbol{xs}) = \mathbf{if}\ \boldsymbol{x} \leqslant p\ \mathbf{then}\ \mathit{partl}\ p\ (ys +\!\!+ [\boldsymbol{x}], zs, \boldsymbol{xs})\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathit{partl}\ p\ (ys, zs +\!\!+ [\boldsymbol{x}], \boldsymbol{xs})\quad .\end{aligned}$$

It might aid our understanding if we note that, if we start *partl* with initial value $([\,], [\,], xs)$ we have the invariant that ys contains elements that are at most p, and elements in zs are larger than p. The calculations below, however, do not rely on this observation. [7]

Our wish is to construct a variant of *partl* that works on arrays. That is, when the array contains $ys +\!\!+ zs +\!\!+ \boldsymbol{xs}$, the three inputs to *partl* in a consecutive segment, when the derived program finishes its work we wish to have $ys +\!\!+ \boldsymbol{us} +\!\!+ zs +\!\!+ \boldsymbol{vs}$, the output of *partl*, stored consecutively in the array.

This brings us to the second issue: *partition*, and therefore *partl*, are stable (that is, elements in each partition retain their original order), which is a strong requirement for array-based partitioning. It is costly to mutate $ys +\!\!+ zs +\!\!+ \boldsymbol{xs}$ into $ys +\!\!+ \boldsymbol{us} +\!\!+ zs +\!\!+ \boldsymbol{vs}$, since it demands that we retain the order of elements in zs while inserting elements of $\boldsymbol{us}$. For sorting we do not need such a strong postcondition. It is sufficient, and can be done more efficiently, to mutate $ys +\!\!+ zs +\!\!+ \boldsymbol{xs}$ into $ys +\!\!+ \boldsymbol{us} +\!\!+ ws$, where ws is some permutation of $zs +\!\!+ \boldsymbol{vs}$. It is handy allowing non-determinism: we introduce a *perm* in our specification, indicating that we do not care about the order of elements in ws.

Define $\mathit{second} :: \mathsf{Monad}\ m \Rightarrow (b \to m\ c) \to (a, b) \to m\ (a, c)$, which applies a monadic function to the second component of a tuple:

$$\mathit{second}\ f\ (x, y) = f\ y \gg\!= \lambda y' \to \{(x, y')\}\quad .$$

Our new wish is to construct an array counterpart of $\mathit{second}\ \mathit{perm} \cdot \mathit{partl}\ p$. Let the function be

$$\begin{aligned}&\mathit{ipartl} :: (\mathsf{MonadPlus}\ m, \mathsf{MonadArr\ Elm}\ m) \Rightarrow\\ &\quad \mathsf{Elm} \to \mathsf{Int} \to (\mathsf{Nat} \times \mathsf{Nat} \times \mathsf{Nat}) \to m\ (\mathsf{Nat} \times \mathsf{Nat})\quad .\end{aligned}$$

[7] It might be worth noting that *partl* causes a space leak in Haskell, since the accumulators become thunks that increase in size as the input list is traversed. It does not matter here since *partl* merely serves as a specification of *ipartl*.

The intention is that in a call $ipartl\ p\ i\ (ny, nz, nx)$, p is the pivot, i the index where $ys +\!\!+ zs +\!\!+ \boldsymbol{xs}$ is stored in the array, and ny, nz, nx respectively the lengths of ys, zs, and xs. A specification of $ipartl$ is:

$$writeList\ i\ (ys +\!\!+ zs +\!\!+ \boldsymbol{xs}) \gg ipartl\ p\ i\ (\#ys, \#zs, \#\boldsymbol{xs}) \subseteq second\ perm\ (partl\ p\ (ys, zs, \boldsymbol{xs})) \gg\!= write2L\ i \quad .$$

That is, under assumption that $ys +\!\!+ zs +\!\!+ \boldsymbol{xs}$ is stored in the array starting from index i (initialised by $writeList$), $ipartl$ computes $partl\ p\ (ys, zs, \boldsymbol{xs})$, possibly permuting the second partition. The resulting two partitions are still stored in the array starting from i, and their lengths are returned.

Derivation We start with fusing $second\ perm$ into $partl$, that is, to construct $partl'\ p \mathrel{\dot\subseteq} second\ perm \cdot partl\ p$.[8] If we discover an inductive definition of $partl'$, it can then be used to construct an inductive definition of $ipartl$. With some routine calculation we get:

$$\begin{aligned}
&partl' :: \mathsf{MonadPlus}\ m \Rightarrow \mathsf{Elm} \to (\mathsf{List\ Elm})^3 \to m\ (\mathsf{List\ Elm} \times \mathsf{List\ Elm}) \\
&partl'\ p\ (ys, zs, [\,]) \quad\ = \{(ys, zs)\} \\
&partl'\ p\ (ys, zs, \boldsymbol{x} : \boldsymbol{xs}) = \\
&\quad \textbf{if}\ \boldsymbol{x} \leqslant p\ \textbf{then}\ perm\ zs \gg\!= \lambda zs' \to partl'\ p\ (ys +\!\!+ [\boldsymbol{x}], zs', \boldsymbol{xs}) \\
&\qquad\quad \textbf{else}\ \ perm\ (zs +\!\!+ [\boldsymbol{x}]) \gg\!= \lambda zs' \to partl'\ p\ (ys, zs', \boldsymbol{xs}) \quad .
\end{aligned}$$

For an intuitive explanation, rather than permuting the second list zs after computing $partl$, we can also permute zs in $partl'$ before every recursive call.

The specification of $ipartl$ now becomes

ipartl_spec

$$writeList\ i\ (ys +\!\!+ zs +\!\!+ \boldsymbol{xs}) \gg ipartl\ p\ i\ (\#ys, \#zs, \#\boldsymbol{xs}) \subseteq partl'\ p\ (ys, zs, \boldsymbol{xs}) \gg\!= write2L\ i \quad . \tag{10}$$

ultimately trying to find ipartl that satisfies this, using this def of partl'

To calculate $ipartl$, we start with the right-hand side of $(\subseteq)$, since it contains more information to work with. We try to push $write2L$ leftwards until the expression has the form $writeList\ i\ (ys +\!\!+ zs +\!\!+ xs) \gg ...$, thereby constructing $ipartl$. This is similar to that, in imperative program calculation, we *work backwards from the postcondition* to construct a program that works under the given precondition [6].

We intend to construct $ipartl$ by induction on xs. For $xs{:=}[\,]$, we get $ipartl\ p\ i\ (ny, nz, 0) = \{(ny, nz)\}$. For the case $x : xs$, assume that the specification is met for xs. Just for making the calculation shorter, we refactor $partl'$, lifting the recursive calls and turning the main body into an auxiliary function:

$$\begin{aligned}
&partl'\ p\ (ys, zs, \boldsymbol{x} : \boldsymbol{xs}) = dispatch\ \boldsymbol{x}\ p\ (ys, zs, \boldsymbol{xs}) \gg\!= partl'\ p \quad , \\
&\quad \textbf{where}\ dispatch\ \boldsymbol{x}\ p\ (ys, zs, \boldsymbol{xs}) = \\
&\qquad \textbf{if}\ \boldsymbol{x} \leqslant p\ \textbf{then}\ perm\ zs \gg\!= \lambda zs' \to \{(ys +\!\!+ [\boldsymbol{x}], zs', \boldsymbol{xs})\} \\
&\qquad\quad \textbf{else}\ \ perm\ (zs +\!\!+ [\boldsymbol{x}]) \gg\!= \lambda zs' \to \{(ys, zs', \boldsymbol{xs})\} \quad .
\end{aligned}$$

[8] We will discover a stronger specification $partl'\ p \mathrel{\dot\subseteq} snd3\ perm \setminus (second\ perm \cdot partl\ p)$, where $snd3\ f\ (x, y, z) = f\ y \gg\!= \lambda y' \to \{(x, y', z)\}$. We omit the details.

We calculate:

$$
\begin{array}{ll}
 & \mathit{partl}'\ p\ (ys, zs, \boldsymbol{x} : \boldsymbol{xs}) \gg\!= \mathit{write2L}\ i \\
= & \quad \{\ \text{definition of } \mathit{partl}'\ \} \\
 & (\mathit{dispatch}\ \boldsymbol{x}\ p\ (ys, zs, \boldsymbol{xs}) \gg\!= \mathit{partl}'\ p) \gg\!= \mathit{write2L}\ i \\
\supseteq & \quad \{\ \text{monad laws, inductive assumption}\ \} \\
 & (\mathit{dispatch}\ \boldsymbol{x}\ p\ (ys, zs, \boldsymbol{xs}) \gg\!= \mathit{write3L}\ i) \gg\!= \mathit{ipartl}\ p\ i \\
= & \quad \{\ \text{by (9), monad laws}\ \} \\
 & \mathit{dispatch}\ \boldsymbol{x}\ p\ (ys, zs, \boldsymbol{xs}) \gg\!= \lambda(ys', zs', \boldsymbol{xs}) \to \\
 & \mathit{writeList}\ i\ (ys' +\!\!+ zs') \gg \mathit{writeList}\ (i + \#(ys' +\!\!+ zs'))\ \boldsymbol{xs} \gg \\
 & \mathit{ipartl}\ p\ i\ (\#ys', \#zs', \#\boldsymbol{xs}) \\
= & \quad \{\ \mathit{perm} \text{ preserves length, commutativity}\ \} \\
 & \mathit{writeList}\ (i + \#ys + \#zs + 1)\ \boldsymbol{xs} \gg \\
 & \mathit{dispatch}\ \boldsymbol{x}\ p\ (ys, zs, \boldsymbol{xs}) \gg\!= \lambda(ys', zs', \boldsymbol{xs}) \to \\
 & \mathit{writeList}\ i\ (ys' +\!\!+ zs') \gg \\
 & \mathit{ipartl}\ p\ i\ (\#ys', \#zs', \#\boldsymbol{xs}) \\
= & \quad \{\ \text{definition of } \mathit{dispatch}\text{, function calls distribute into } \textbf{if}\ \} \\
 & \mathit{writeList}\ (i + \#ys + \#zs + 1)\ \boldsymbol{xs} \gg \\
 & \textbf{if}\ \boldsymbol{x} \leqslant p\ \textbf{then}\ \mathit{perm}\ zs \gg\!= \lambda zs' \to \mathit{writeList}\ i\ (ys +\!\!+ [\boldsymbol{x}] +\!\!+ zs') \gg \\
 & \qquad\qquad \mathit{ipartl}\ p\ i\ (\#ys + 1, \#zs', \#\boldsymbol{xs}) \\
 & \qquad\ \textbf{else}\ \ \mathit{perm}\ (zs +\!\!+ [\boldsymbol{x}]) \gg\!= \lambda zs' \to \mathit{writeList}\ i\ (ys +\!\!+ zs') \gg \\
 & \qquad\qquad \mathit{ipartl}\ p\ i\ (\#ys, \#zs', \#\boldsymbol{xs})\ \ .
\end{array}
$$

We pause here to see what has happened: we have constructed a precondition $\mathit{writeList}\ (i + \#ys + \#zs + 1)\ \boldsymbol{xs}$, which is part of the desired precondition: $\mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ (\boldsymbol{x} : \boldsymbol{xs}))$. To recover the latter precondition, we will try to turn both branches of **if** into the form $\mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ [\boldsymbol{x}]) \gg\!= \ldots$. That is, we try to construct, in both branches, some code that executes under the precondition $\mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ [\boldsymbol{x}])$ — that the code generates the correct result is guaranteed by the refinement relation.

It is easier for the second branch, where we can simply refine *perm* to $\{\cdot\}$:

$$
\begin{array}{ll}
 & \mathit{perm}\ (zs +\!\!+ [\boldsymbol{x}]) \gg\!= \lambda zs' \to \mathit{writeList}\ i\ (ys +\!\!+ zs') \gg \\
 & \mathit{ipartl}\ p\ i\ (\#ys, \#zs', \#\boldsymbol{xs}) \\
\supseteq & \quad \{\ \text{since } \{xs\} \subseteq \mathit{perm}\ xs\ \} \\
 & \mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ [\boldsymbol{x}]) \gg \mathit{ipartl}\ p\ i\ (\#ys, \#zs + 1, \#\boldsymbol{xs})\ \ .
\end{array}
$$

ipartl_spec_lemma1

For the first branch, we focus on its first line:

$$
\begin{array}{ll}
 & \mathit{perm}\ zs \gg\!= \lambda zs' \to \mathit{writeList}\ i\ (ys +\!\!+ [\boldsymbol{x}] +\!\!+ zs') \\
= & \quad \{\ \text{by (9), commutativity}\ \} \\
 & \mathit{writeList}\ i\ ys \gg \mathit{perm}\ zs \gg\!= \lambda zs' \to \mathit{writeList}\ (i + \#ys)\ ([\boldsymbol{x}] +\!\!+ zs') \\
\supseteq & \quad \{\ \text{introduce } \mathit{swap}\text{, see below}\ \} \\
 & \mathit{writeList}\ i\ ys \gg \mathit{writeList}\ (i + \#ys)\ (zs +\!\!+ [\boldsymbol{x}]) \gg \\
 & \mathit{swap}\ (i + \#ys)\ (i + \#ys + \#zs) \\
= & \quad \{\ \text{by (9)}\ \} \\
 & \mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ [\boldsymbol{x}]) \gg \mathit{swap}\ (i + \#ys)\ (i + \#ys + \#zs)\ \ .
\end{array}
$$

ipartl_spec_lemma2

Here we explain the last two steps. Operationally speaking, given an array containing $ys +\!\!+ zs +\!\!+ [\boldsymbol{x}]$ (the precondition we wanted, initialized by the *writeList* in the last line), how do we mutate it to $ys +\!\!+ [\boldsymbol{x}] +\!\!+ zs'$ (postcondition specified by the *writeList* in the first line), where zs' is a permutation of zs? We may do so by swapping $\boldsymbol{x}$ with the leftmost element of zs, which is what we did in the second step. Formally, we used the property:

ipartl_spec_lemma3

$$\begin{aligned} &\mathit{perm}\ zs \gg\!= \lambda zs' \to \mathit{writeList}\ i\ ([\boldsymbol{x}] +\!\!+ zs') \supseteq \\ &\qquad \mathit{writeList}\ i\ (zs +\!\!+ [\boldsymbol{x}]) \gg \mathit{swap}\ i\ (i + \#zs)\ \ . \end{aligned} \tag{11}$$

Now that both branches are refined to code with precondition $\mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ [\boldsymbol{x}])$, we go back to the main derivation:

$$\begin{aligned} &\mathit{writeList}\ (i + \#ys + \#zs + 1)\ \boldsymbol{xs} \gg \\ &\mathbf{if}\ \boldsymbol{x} \leqslant p\ \mathbf{then}\ \mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ [\boldsymbol{x}]) \gg \\ &\qquad\qquad \mathit{swap}\ (i + \#ys)\ (i + \#ys + \#zs) \gg \\ &\qquad\qquad \mathit{ipartl}\ p\ i\ (\#ys + 1, \#zs, \#\boldsymbol{xs}) \\ &\qquad \mathbf{else}\ \ \mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ [\boldsymbol{x}]) \gg \\ &\qquad\qquad \mathit{ipartl}\ p\ i\ (\#ys, \#zs + 1, \#\boldsymbol{xs}) \\ =\ &\ \{\ \text{distributivity of } \mathbf{if},\ (9)\ \} \\ &\mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ (\boldsymbol{x} : \boldsymbol{xs})) \gg \\ &\mathbf{if}\ \boldsymbol{x} \leqslant p\ \mathbf{then}\ \mathit{swap}\ (i + \#ys)\ (i + \#ys + \#zs) \gg \\ &\qquad\qquad \mathit{ipartl}\ p\ i\ (\#ys + 1, \#zs, \#\boldsymbol{xs}) \\ &\qquad \mathbf{else}\ \ \mathit{ipartl}\ p\ i\ (\#ys, \#zs + 1, \#\boldsymbol{xs}) \\ =\ &\ \{\ \textbf{write-read} \text{ and definition of } \mathit{writeList}\ \} \\ &\mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ (\boldsymbol{x} : \boldsymbol{xs})) \gg \\ &\mathit{read}\ (i + \#ys + \#zs) \gg\!= \lambda \boldsymbol{x} \to \\ &\mathbf{if}\ \boldsymbol{x} \leqslant p\ \mathbf{then}\ \mathit{swap}\ (i + \#ys)\ (i + \#ys + \#zs) \gg \\ &\qquad\qquad \mathit{ipartl}\ p\ i\ (\#ys + 1, \#zs, \#\boldsymbol{xs}) \\ &\qquad \mathbf{else}\ \ \mathit{ipartl}\ p\ i\ (\#ys, \#zs + 1, \#\boldsymbol{xs})\ \ . \end{aligned}$$

We have thus established the precondition $\mathit{writeList}\ i\ (ys +\!\!+ zs +\!\!+ (\boldsymbol{x} : \boldsymbol{xs}))$. In summary, we have derived:

$$\begin{aligned} &\mathit{ipartl} :: \mathsf{MonadArr\ Elm}\ m \Rightarrow \mathsf{Elm} \to \mathsf{Int} \to (\mathsf{Int} \times \mathsf{Int} \times \mathsf{Int}) \to m\ (\mathsf{Int} \times \mathsf{Int}) \\ &\mathit{ipartl}\ p\ i\ (ny, nz, 0) \qquad = \{(ny, nz)\} \\ &\mathit{ipartl}\ p\ i\ (ny, nz, 1 + k) = \\ &\quad \mathit{read}\ (i + ny + nz) \gg\!= \lambda x \to \\ &\quad \mathbf{if}\ x \leqslant p\ \mathbf{then}\ \mathit{swap}\ (i + ny)\ (i + ny + nz) \gg \mathit{ipartl}\ p\ i\ (ny + 1, nz, k) \\ &\qquad\qquad \mathbf{else}\ \ \mathit{ipartl}\ p\ i\ (ny, nz + 1, k)\ \ . \end{aligned}$$

5.3 Sorting an Array

Now that we have *ipartl* derived, the rest of the work is to install it into quicksort. We intend to derive $\mathit{iqsort} :: \mathsf{MonadArr\ Elm}\ m \Rightarrow \mathsf{Int} \to \mathsf{Nat} \to m\ ()$ such that

isort i n sorts the n elements in the array starting from index i. We can give it a formal specification:

iqsort_spec

$$writeList\ i\ xs \gg iqsort\ i\ (\#xs)\ \subseteq\ slowsort\ xs \gg\!= writeList\ i\ \ . \tag{12}$$

That is, when *iqsort i* is run from a state initialised by *writeList i xs*, it should behave the same as *slowsort xs* $\gg\!=$ *writeList i*.

The function *iqsort* can be constructed by induction on the length of the input list. For the case $xs := p : xs$, we start from the left-hand side *slowsort* $(p : xs) \gg\!=$ *writeList i* and attempt to transform it to *writeList i* $(p : xs) \gg$ ..., thereby construct *iqsort*. We present only the hightlights of the derivation. Firstly, *slowsort* $(p : xs) \gg\!=$ *writeList i* can be transformed to:

iqsort_spec_lemma1

$$\begin{array}{l} partl'\ p\ ([\,],[\,],xs) \gg\!= \lambda(ys,zs) \rightarrow \\ perm\ ys \gg\!= \lambda ys' \rightarrow writeList\ i\ (ys' +\!\!+ [p] +\!\!+ zs) \gg \\ iqsort\ i\ (\#ys) \gg iqsort\ (i + \#ys + 1)\ (\#zs)\ \ . \end{array}$$

For that to work, we introduced two *perm* to permute both partitions generated by *partition*. We can do so because *perm* $\gg\!\!>$ *perm* $=$ *perm* and thus *perm* $\gg\!\!>$ *slowsort* $=$ *slowsort*. The term *perm zs* was combined with *partition p*, yielding *partl'* p, while *perm ys* will be needed later. We also needed (9) to split *writeList i* $(ys' +\!\!+ [x] +\!\!+ zs')$ into two parts. Assuming that (12) has been met for lists shorter than xs, two subexpressions are folded back to *iqsort*.

Now that we have introduced *partl'*, the next goal is to embed *ipartl*. The status of the array before the two calls to *iqsort* is given by *writeList i* $(ys' +\!\!+ [p] +\!\!+ zs)$. That is, $ys' +\!\!+ [p] +\!\!+ zs$ is stored in the array from index i, where ys' is a permutation of ys. The postcondition of *ipartl*, according to the specification (10), ends up with ys and zs stored consecutively. To connect the two conditions, we use a lemma that is dual to (11):

iqsort_spec_lemma2

$$\begin{array}{l} perm\ ys \gg\!= \lambda ys' \rightarrow writeList\ i\ (ys' +\!\!+ [p]) \supseteq \\ \qquad writeList\ i\ ([p] +\!\!+ ys) \gg swap\ i\ (i + \#ys)\ \ . \end{array} \tag{13}$$

This is what the typical quicksort algorithm does: swapping the pivot with the last element of ys, and (13) says that it is valid because that is one of the many permutations of ys. With (13) and (10), the specification can be refined to:

iqsort_spec_lemma3

$$\begin{array}{l} writeList\ i\ (p : xs) \gg \\ ipartl\ p\ (i+1)\ (0,0,\#xs) \gg\!= \lambda(ny,nz) \rightarrow swap\ i\ (i + ny) \gg \\ iqsort\ i\ (\#ys) \gg iqsort\ (i + \#ys + 1)\ (\#zs)\ \ . \end{array}$$

In summary, we have derived:

$$\begin{array}{l} iqsort :: \mathsf{MonadArr\ Elm}\ m \Rightarrow \mathsf{Int} \rightarrow \mathsf{Nat} \rightarrow m\ () \\ iqsort\ i\ 0 = \{()\} \\ iqsort\ i\ n = read\ i \gg\!= \lambda p \rightarrow \\ \qquad\qquad ipartl\ p\ (i+1)\ (0,0,n-1) \gg\!= \lambda(ny,nz) \rightarrow \\ \qquad\qquad swap\ i\ (i + ny) \gg \\ \qquad\qquad iqsort\ i\ ny \gg iqsort\ (i + ny + 1)\ nz\ \ . \end{array}$$

6 Conclusions

From a specification of sorting using the non-determinism monad, we have derived a pure quicksort for lists and a state-monadic quicksort for arrays. We hope to demonstrate that the monadic style is a good choice as a calculus for program derivation that involves non-determinism. One may perform the derivation in pointwise style, and deploy techniques that functional programmers have familiarised themselves with, such as pattern matching and induction on structures or on sizes. When preferred, one can also work in point-free style with $(\gg\!\!=)$. Programs having other effects can be naturally incorporated into this framework. The way we derive stateful programs echos how we, in Dijkstra's style, reason backwards from the postcondition.

A final note: $(\gg\!\!=)$ and $(\dot{\subseteq})$ naturally induce the notion of (left) factor, $(\backslash) :: (a \to m\ b) \to (a \to m\ c) \to b \to m\ c$, defined by the Galois connection:

$$f \gg\!\!= g\ \dot{\subseteq}\ h\ \equiv\ g\ \dot{\subseteq}\ f \backslash h\ \ .$$

Let $h :: a \to m\ c$ be a monadic specification, and $f :: a \to m\ b$ performs the computation half way, then $f \backslash h$ is the most non-deterministic (least constrained) monadic program that, when ran after the postcondition set up by f, still meets the result specified by h. With $(\backslash)$, *ipartl* and *iqsort* can be specified by:

$$\begin{aligned} &\mathit{ipartl}\ p\ i \dot{\subseteq} \mathit{write3L}\ i \backslash ((\mathit{second}\ \mathit{perm} \cdot \mathit{partl}\ p) \gg\!\!= \mathit{write2L}\ i)\ \ , \\ &\mathit{iqsort}\ i\ \ \dot{\subseteq} \mathit{writeL}\ i \backslash (\mathit{slowsort} \gg\!\!= \mathit{writeList}\ i)\ \ . \end{aligned}$$

In relational calculus, the *factor* is an important operator that is often associated with weakest precondition. We unfortunately cannot cover it due to space constraints.

Acknowledgements The authors would like to thank Jeremy Gibbons for the valuable discussions during development of this work.

References

1. Backhouse, R.C., de Bruin, P.J., Malcolm, G., Voermans, E., van der Woude, J.: Relational catamorphisms. In: Möller, B. (ed.) Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs. pp. 287–318. Elsevier Science Publishers (1991)
2. Bentley, J.L.: Programming Pearls, Second Edition. Addison-Wesley (2000)
3. Bird, R.S.: Functional algorithm design. Science of Computer Programming **26**, 15–31 (1996)
4. Bird, R.S., de Moor, O.: Algebra of Programming. International Series in Computer Science, Prentice Hall (1997)
5. Bird, R.S., Rabe, F.: How to calculate with nondeterministic functions. In: Hutton, G. (ed.) Mathematics of Program Construction. pp. 138–154. Springer (2019)
6. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)

7. Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: Danvy, O. (ed.) International Conference on Functional Programming. pp. 2–14. ACM Press (2011)
8. Gill, A., Kmett, E.: The monad transformer library. https://hackage.haskell.org/package/mtl (2014)
9. Hoare, C.A.R.: Algorithm 63: Partition. Communications of the ACM **4**(7), 321 (1961)
10. Kiselyov, O.: How to restrict a monad without breaking it: the winding road to the Set monad. http://okmij.org/ftp/Haskell/set-monad.html (July 2013)
11. Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: Reppy, J.H. (ed.) Symposium on Haskell. pp. 94–105. ACM Press (2015)
12. Moggi, E.: Computational lambda-calculus and monads. In: Parikh, R. (ed.) Logic in Computer Science. pp. 14–23. IEEE Computer Society Press (1989)
13. de Moor, O., Gibbons, J.: Pointwise relational programming. In: Rus, T. (ed.) Algebraic Methodology and Software Technology. pp. 371–390. No. 1816 in Lecture Notes in Computer Science, Springer-Verlag (2000)
14. Pauwels, K., Schrijvers, T., Mu, S.C.: Handling local state with global state. In: Hutton, G. (ed.) Mathematics of Program Construction. pp. 18–44. Springer (2019)