Journal of Logical and Algebraic Methods in Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Formal derivation of Greedy algorithms from relational specifications: A tutorial

Yu-Hsi Chiang [a], Shin-Cheng Mu [b,*]

[a] Dep. of Computer Science and Information Engineering, National Taiwan University, Taiwan
[b] Institute of Information Science, Academia Sinica, Taiwan

A R T I C L E I N F O

Article history:
Received 16 June 2015
Received in revised form 23 December 2015
Accepted 23 December 2015
Available online xxxx

Keywords:
Program derivation
Algebra of programming
Galois connection
Dependent type
Machine-aided theorem proving

A B S T R A C T

Many programming tasks can be specified as optimisation problems in which a relation is used to generate all possible solutions, from which we wish to choose an optimal one. A relational operator "shrink", developed by José N. Oliveira, is particularly suitable for constructing greedy algorithms from such specifications. Meanwhile, it has become standard in many sub-fields in programming language that proofs must be machine-verified. This tutorial leads the readers through the development of algebraic derivations of three greedy algorithms, one fold-based and two unfold-based, using AoPA, a library designed for machine-verified relational program calculation.

1. Introduction

Many programming tasks can be described as to look for, among a collection of solution candidates, the maximum/minimum solution under some ordering. To compute integral division $x \div y$, for example, is to find out the largest integer that, when multiplied by y, does not exceed x. Similarly, to solve the minimum coin change problem is to choose, among the lists of coins whose sum is the given amount, the shortest such list.

Since the 90s, the program derivation community has been exploring moving from functions to a relational theory of program calculation [2–4], and expressing such optimisation problems as relations, culminating in the work of Bird and de Moor [5]. More recently, during their investigation of Galois connections, Mu and Oliveira [18,19] proposed a relational operator ($\upharpoonright$), pronounced "shrink". Let S be a relation that nondeterministically maps the input to a possible solution candidate of an optimisation problem, and R a preorder on the solutions. The relation $S \upharpoonright R$, pronounced "S shrunk by R", is a subrelation of S, with some entries removed, such that an input is mapped to a solution only if it is minimum under R. A number of theorems, for example, one that allows ($\upharpoonright R$) to be promoted into S when the latter is a relational fold, were presented to refine the relational specification to a functional program. This formulation is less general than that of Bird and de Moor [5] but, being simpler and not involving set-valued functions, particularly suitable for constructing greedy algorithms.

Meanwhile, the programming language community in recent years has been moving toward more reliance on machine-aided program verification. Complex theorems are considered "proved" only if the proof is verified by a theorem prover or

* Corresponding author.
E-mail address: scm@iis.sinica.edu.tw (S.-C. Mu).

http://dx.doi.org/10.1016/j.jlamp.2015.12.003
2352-2208/© 2016 Elsevier Inc. All rights reserved.

a proof assistant, and a paper published in a top-tier conference is often supposed to have its theorems formally verified by machine.

To meet the demand, AoPA (Algebra of Programming in Agda, Mu et al. [17]) is a library, built in the dependently typed functional language Agda, that allows one to develop relational proofs in a style similar to that of Bird and de Moor [5]. Agda [21] is both a dependently typed programming language and a proof assistant, adopting a Martin-Löf style type system. Unlike theorem provers such as Coq, in which the user issues commands that instruct the computer to apply logic or rewriting rules, Agda exploits Curry–Howard isomorphism and allows the users to directly write down proof terms as programs. Partially completed programs (proofs) may contain holes, whose type the programmer may enquiry. The programmer tries to fill the hole with a term meeting the demanded type, during the process more holes may be spawn. We find this interface suitable for developing calculational proofs, where a calculation in progress is carried out step-by-step.

In AoPA, elements of the relational theory of programs are modelled in the type system of Agda. Calculations are expressed as program terms that, if type-checked, is guaranteed to be correct. In the end of a calculation one may extract a functional program that is proved to respect the relational specification.

This expository paper is a tutorial on formal relational program derivation of greedy algorithms in AoPA, using José N. Oliveira's shrink operator. The target readers are those who have some experience in relational or functional program calculation and are in need of tools to verify the proofs. The theory of relational program calculation has been summarised by Mu and Oliveira [19], while the textbook of Bird and de Moor [5] presents the complete theory from a category-theoretical point of view. This paper, complementarily, discusses issues that arise when the theories are embedded in a type theory and applied to construct executable programs, including how relations and relational operators are encoded, how some definitions in the previous work become theorems in Agda, and how proofs of termination are used to extract executable programs from specifications. This paper also documents various features of AoPA that were implemented after the publication of Mu et al. [17], including generic datatypes and folds, and new greedy theorems that apply to unfolds and hylomorphisms.

This paper is dedicated to the 60th birthday of José N. Oliveira. The second author of this paper had the pleasure of working with him during around 2010 to 2012, exploring the link between relational program derivation and Galois connections. While working with him, the second author benefited not only from José's rich knowledge and intuition to mathematics and programming, which proved to be always correct, but also from his gentle and friendly attitude to junior researchers, and his belief that teaching a rigorous approach to programming to students is not only possible but rewarding. José insisted that authors of academic papers should be listed in alphabetical order, a convention this paper also follows.

Running examples. We will use three running examples throughout the paper. The first example is the function takeWhile, which takes a predicate p and a list, and returns the longest prefix whose elements all satisfy the predicate. The function takeWhile can be specified relationally as

$$\begin{aligned}&\mathsf{takeWhile} : \forall \{\mathsf{A}\} \to (\mathsf{A} \to \mathsf{Bool}) \to (\mathsf{List\ A} \leftarrow \mathsf{List\ A})\\ &\mathsf{takeWhile\ p} = ((\mathsf{all\ p})\ ¿' \circ (\preccurlyeq)) \upharpoonright (\succcurlyeq) \quad ,\end{aligned}$$

where $(\preccurlyeq)$ denotes the prefix relation: $\mathsf{xs} \preccurlyeq \mathsf{ys}$ if xs is a prefix of ys. The relation $(\mathsf{all\ p})\ ¿' \circ (\preccurlyeq)$ maps the input list to one of its prefixes, while making sure that every element of the latter satisfies p. The part $(\upharpoonright (\succcurlyeq))$ denotes that we want the longest such prefix. The notations will be explained in detail in the sections to follow.

The second example is the function take, such that take (n , xs) returns the first n elements of the list xs and, if xs contains less than n elements, returns xs itself. A relational specification of take will be given in Section 4.2.

The final example, inspired by Ko and Gibbons [14], is the minimum coin change problem: how to make change for a given amount, using a fixed set of denomination (1p, 2p, 5p, and 10p), such that the number of coins used is minimum? To solve the problem using a proof assistant, it helps to explicitly enumerate the set of allowed coins as a datatype:

```
data Coin : Set where
  1p : Coin;   2p  : Coin
  5p : Coin;   10p : Coin  .
```

The problem may then be specified by:

$$\begin{aligned}&\mathsf{coin\text{-}change} : \mathsf{List\ Coin} \leftarrow \mathbb{N}\\ &\mathsf{coin\text{-}change} = (\mathsf{sum} \circ \mathsf{ordered}\ ¿)^{\smile} \upharpoonright (\leq_l) \quad ,\end{aligned}$$

where sum computes the sum of a list of coins, while ordered ¿ ensures that the list is sorted in descending values of coins. The operator $\_^{\smile}$ stands for relational converse, a concept similar to inverse functions. Thus $(\mathsf{sum} \circ \mathsf{ordered}\ ¿)^{\smile}$ is a relation that maps the input, the given amount, to a sorted list of coins that sums up to it, from which $(\upharpoonright (\leq_l))$ chooses a shortest one — $(\leq_l)$ compares lists by their lengths. From the specification, we wish to derive a greedy algorithm. It will be explained in Section 7.3 why it is important that coin-change generates output sorted output.

Outline. In Section 2 we give a crash course in Agda, including the language itself and its interactive interface. We review the basic elements of relational program calculation and previous results on Galois connection respectively in Section 3