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

Towards patterns for heaps and imperative lambdas

David A. Naumann

Stevens Institute of Technology, USA

ARTICLE INFO

Article history:
Received 19 June 2015
Received in revised form 7 October 2015
Accepted 17 October 2015
Available online xxxx

Dedicated to José Nuno Oliveira on the occasion of his 60th birthday

ABSTRACT

In functional programming, pointfree relation calculi have been fruitful for general theories of program construction, but for specific applications pointwise expressions can be more convenient and comprehensible. In imperative programming, refinement calculi have been tied to pointwise expression in terms of state variables, with the curious exception of the ubiquitous but invisible heap. To integrate pointwise with pointfree, de Moor and Gibbons [12] extended lambda calculus with non-injective pattern matching interpreted using relations. This article gives a semantics of that language using "ideal relations" between partial orders, and a second semantics using predicate transformers. The second semantics is motivated by its potential use with separation algebra, for pattern matching in programs acting on the heap. Laws including lax beta and eta are proved in these models and a number of open problems are posed.

1. Introduction

An important idea in the mathematics of program construction is to embed the programming language of interest into a richer language with additional features that are useful for writing specifications and for reasoning. Functional programs can be embedded in the calculus of relations, which provides two key benefits: converse functions as specifications and intersection of specifications. An example of the first benefit is parsing. Let $show : Tree \longrightarrow String$ be the function that maps an ordered tree with strings at its leaves to the "inorder" catenation of the leaves. Its converse, $show^o$, is a relation but not a function. One seeks to derive, by algebraic reasoning in the calculus of relations, a total function *parse* such that $show^o \supseteq parse$. See Bird and de Moor [2] for many more examples. Imperative programs can be embedded in a refinement calculus [9,23], by augmenting the language with assumptions and angelic choice, or "specification statements" in some other form. These can be modeled using weakest precondition predicate transformers. An imperative program *prog* satisfies specification *spec* just if $spec \sqsubseteq prog$ where $\sqsubseteq$ is the pointwise order on predicate transformers, and again one seeks to derive *prog* from *spec*.

Many authors have pointed out useful and elegant aspects of the calculus of relations for programming. Relations cater for the development of general theory by facilitating a "point free" style in which algebraic calculation is not encumbered by manipulation of bound variables and substitutions (e.g., see [31]).

Although pointfree style is elegant and effective for development of general theory, it can be awkward and cryptic for developing and expressing specific algorthms. Functional programmers tend to prefer a mix of pointfree and pointwise expressions, "pointwise" meaning the use of variables and other expressions that denote data elements—application rather than composition. Pointwise reasoning involves logical quantifiers and is the norm in imperative program construction. For

E-mail address: naumann@cs.stevens.edu.

http://dx.doi.org/10.1016/j.jlamp.2015.10.008
2352-2208/

example, refinement laws for assignment statements involve conditions on free variables, and specifications are expressed in terms of state variables and formulas with quantifiers.

Conventional pattern matching can help raise the abstraction level in pointwise programs, by directly expressing data structure of interest. Non-injective patterns have been proposed by de Moor and Gibbons [12] as a way to achieve pointwise programming with relations. Imperative programmers draw graphs to express patterns of pointer structure, but their programs are written in impoverished notation that amounts to little more than load and store instructions.

This article contributes to the long term goal of a unified theory of programming in which one may move freely between pointwise or pointfree reasoning as suits the occasion. For example, requirements might be formalized in a transparent pointwise specification that is then transformed to a pointfree equivalent from which an efficient solution is derived by algebraic calculation. A unified theory will also enable effective mixes of functional, imperative, and other styles both in program structure and in reasoning.

This article describes one approach to a programming calculus integrating functional and imperative styles, addressing some aspects of pointwise and pointfree reasoning. Some of the technical results were published in a conference paper by the author [29], from which much of the material is adapted. The introductory sections have been rewritten using different examples. This article provides full details of the main semantic definitions and some results only mentioned sketchily in the conference paper, namely beta and eta laws. We cannot expect beta and eta equalities to hold unrestrictedly, as they fail already in by-value functional languages. Inequational laws are mentioned but not proved in [12] and [29]. Here we prove weak beta and eta laws for both relational and predicate transformer semantics. We also pose several open problems.

Outline. The rest of this article is organized as follows. Section 2 begins with motivation, focusing on higher types and the idea of non-injective patterns. We show by example how non-injective patterns could be used in imperative programming including pointer programs. This idea helps motivate the predicate transformer semantics but is not otherwise developed in this article. Section 2 also surveys related work on alternate approaches to programming calculi integrating pointwise with pointfree and functional with imperative styles.

Section 3 reviews the standard semantics of simply typed lambda calculus in a Cartesian closed category, in particular **Poset**. Section 4 describes the category of ideal relations, motivated by difficulties with semantics in [12]. Section 5 gives our relational semantics. Section 6 gives a simulation connecting relational and functional semantics, and proves the lax beta and eta laws that are our main results for relational semantics. Section 7 gives the predicate transformer model and semantics. Section 8 proves the main results for transformer semantics. Section 9 assesses the work and discusses open problems.

For Section 3 onwards, the reader should be familiar with predicate transformer semantics [9] and with basic category theory including adjunctions and Cartesian closure [15]. Span constructions and lax adjunctions are only mentioned in passing, and "laxity" appears only as an informal term that indicates the weakening of equations to inequations.

2. Motivation and background

Motivation. One attraction of pointfree style is that it facilitates derivation of programs that are "polytypic", i.e., generic in some sense with respect to type constructors [4]. For example, a polynomial functor on a category of data types may have a fixpoint; its values are trees of some form determined by the particular functor. If the element type has a well ordering, one can define the function *repmin* that sends tree t to the tree t' of the same shape but where each leaf of t' is the minimum of the leaves of t. De Moor gives a pointfree derivation of *repmin*, at this level of generality, using type constructions and equational laws that can be interpreted in functions or in relations [10].

Relations can model demonic nondeterminacy [12] or angelic nondeterminacy (as in automata theory and in logic programming), but not both—unless states or data values are replaced by richer structures such as predicates. The present author showed that the algebraic structure needed for the polytypic *repmin* derivation exists in the setting of monotonic predicate transformers [26].

Although the *repmin* problem only involves first order data (trees with primitive, ordered data), the derived solution involves higher order: It traverses the input tree to build a closure that, when applied to a value, builds a tree of the same shape with that value at its leaves. This brings us to a question about how to embed a programming language in a richer calculus for specification and derivation. In the language of categories, taking data types as objects and programs as arrows, the question is what objects to use for arrow types. For each pair B, C of objects, a function space $B \Rightarrow C$ exists as an object in the category **Rel** of binary relations, and indeed as an object in the category of monotonic predicate transformers. But $B \Rightarrow C$ is not the "internal hom" or exponent in **Rel**. There should be some account of what it means to reason with exponents in **Rel** if the derived program is interpreted as a functional one.

Pointfree reasoning is not without its shortcomings. De Moor and Gibbons observe that for many specific programming problems a pointwise formulation is easier to understand. They extend pointwise functional notation to relations by means of non-injective patterns. As a simple example, the following is intended to define a relation that performs an arbitrary rotation of a list:

$$rotate(x +\!\!+ y) = y +\!\!+ x \tag{1}$$

Please cite this article in press as: D.A. Naumann, Towards patterns for heaps and imperative lambdas, Journal of Logical and Algebraic Methods in Programming (2015), http://dx.doi.org/10.1016/j.jlamp.2015.10.008