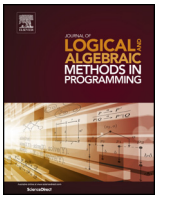
Journal of Logical and Algebraic Methods in Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Join inverse categories and reversible recursion ☆

Robin Kaarsgaard *, Holger Bock Axelsen, Robert Glück

DIKU, Department of Computer Science, University of Copenhagen, Denmark

ARTICLE INFO

Article history:
Received 31 January 2016
Received in revised form 14 August 2016
Accepted 16 August 2016
Available online xxxx

Keywords:
Reversible computing
Recursion
Categorical semantics
Enriched category theory

ABSTRACT

Recently, a number of reversible functional programming languages have been proposed. Common to several of these is the assumption of totality, a property that is not necessarily desirable, and certainly not required in order to guarantee reversibility. In a categorical setting, however, faithfully capturing partiality requires handling it as additional structure. Recently, Giles studied inverse categories as a model of partial reversible (functional) programming. In this paper, we show how additionally assuming the existence of countable joins on such inverse categories leads to a number of properties that are desirable when modeling reversible functional programming, notably morphism schemes for reversible recursion, a †-trace, and algebraic ω-compactness. This gives a categorical account of reversible recursion, and, for the latter, provides an answer to the problem posed by Giles regarding the formulation of recursive data types at the inverse category level.

1. Introduction

Reversible computing, that is, the study of computations that exhibit both forward and backward determinism, originally grew out of the thermodynamics of computation. Landauer's principle states that computations performed by some physical system (thermodynamically) dissipate heat when information is erased, but that no dissipation is entailed by information-preserving computations [3]. This has motivated a long study of diverse reversible computation models, such as logic circuits [4], Turing machines [5,6], and many forms of restricted automata models [7,8]. Reversibility concepts are important in quantum computing, but are increasingly seen to be of interest in other areas as well, including high-performance computing [9], process calculi [10], and even robotics [11,12].

In this paper we concern ourselves with the categorical underpinnings of reversible functional programming languages. At the programming language level, reversible languages exhibit interesting program properties, such as easy program inversion [13]. Now, most reversible languages are stateful, giving them a fairly straightforward semantic interpretation [14]. While functional programs are usually easier to reason about at the meta-level, they do not have the concept of state that imperative languages do, making their semantics interesting objects of study.

Further, many reversible functional programming languages (such as Theseus [15] and the Π-family of combinator calculi [16]) come equipped with a tacit assumption of totality, a property that is neither required [6] nor necessarily desirable as far as guaranteeing reversibility is concerned. Shedding ourselves of the "tyranny of totality," however, requires us to handle partiality explicitly as additional categorical structure.

☆ This is an extended version of an abstract presented at NWPT 2015 [1] and a paper presented at FoSSaCS 2016 [2], elaborated with full proofs, additional examples, and more comprehensive background.

* Corresponding author.
E-mail addresses: robin@di.ku.dk (R. Kaarsgaard), funkstar@di.ku.dk (H.B. Axelsen), glueck@di.ku.dk (R. Glück).

http://dx.doi.org/10.1016/j.jlamp.2016.08.003
2352-2208/

One approach which does precisely that is inverse categories, as studied by Cockett and Lack [17] as a specialization of restriction categories, which have recently been suggested and developed by Giles [18] as models of reversible (functional) programming. In this paper, we will argue that assuming ever slightly more structure on these inverse categories, namely the presence of *countable joins* of parallel morphisms [19], gives rise to a number of additional properties useful for modeling reversible functional programming. Notably, we obtain two different notions of reversible recursion (exemplified in the two different reversible languages RFUN and Theseus), and an account of recursive data types (via algebraic ω-compactness with respect to structure-preserving functors), which are not present in the general case. This is done by adopting two different, but complementary, views on inverse categories with countable joins as enriched categories – as **DCPO**-categories, and as (specifically Σ**Mon**-enriched) strong unique decomposition categories [20,21].

Overview. We give a brief introduction to reversible functional programming, specifically to the languages of RFUN [22] and Theseus [15], in Section 2, and present the necessary background on restriction and inverse categories in Section 3. In Section 4 we show that inverse categories with countable joins are **DCPO**-enriched, which allows us to demonstrate the existence of (reversible!) fixed points of both morphism schemes and structure-preserving functors. In Section 5 we show that inverse categories with countable joins and a join-preserving disjointness tensor are (strong) unique decomposition categories equipped with a uniform †-trace. Section 6 gives conclusions and directions for future work.

2. On reversible functional programming

In this section, we give a brief introduction to reversible functional programming, specifically to the languages of RFUN and Theseus. For more comprehensive accounts of these languages, including syntax, semantics, program inversion, further examples, and so on, see [22] respectively [15].

Reversible programming deals with the construction of programs that are deterministic not just in the forward direction (as any other deterministic program), but also in the backward direction. A central consequence of this property is that well-formed programs must have both uniquely defined forward and backward semantics, with backward semantics given either directly or indirectly (*e.g.*, as is often done, by providing a textual translation of terms into terms which carry their inverse semantics; this approach is related to program inversion [23,24]). In the case of reversible functional programming, reversibility is accomplished by guaranteeing local (forward and backward) determinism of evaluation – which, in turn, leads to global (forward and backward) determinism. Though reversible functions are injective [6], injectivity itself (a *global* property) is not enough to guarantee reversibility (a *local* property) – specifically, locally reversible control structures are necessary [22].

One such reversible functional programming language is RFUN, developed in recent years by Yokoyama, Axelsen, and Glück [22]. RFUN is an untyped language that uses Lisp-style symbols and constructors for data representation. Programs in RFUN are first-order functions, in which bound variables must be linearly used (though patterns are not required to be exhaustive). To account for the fact that data duplication *can* be performed reversibly, a *duplication-equality* operator [25], defined as follows, is used:

$$\lfloor \langle x \rangle \rfloor = \langle x, x \rangle$$
$$\lfloor \langle x, y \rangle \rfloor = \begin{cases} \langle x \rangle & \text{if } x = y \\ \langle x, y \rangle & \text{otherwise} \end{cases}$$

In the first case, the application of $\lfloor \cdot \rfloor$ to the unary tuple $\langle x \rangle$ yields the binary tuple $\langle x, x \rangle$, that is, the value x is duplicated. In the second case, when $x = y$, the application to $\langle x, y \rangle$ joins two identical values into $\langle x \rangle$; otherwise, the two values are returned unchanged (two different values cannot have been obtained by duplication of one value). Using an explicit operator simplifies reverse computation because the duplication of a value in one direction requires an equality check in the other direction, and *vice versa*. Instead of using a variable twice to duplicate a value, the duplication is made explicit. The operator is self-inverse, *e.g.*, $\lfloor \lfloor \langle x \rangle \rfloor \rfloor = \langle x \rangle$ and $\lfloor \lfloor \langle x, y \rangle \rfloor \rfloor = \langle x, y \rangle$.

The only control structure available in RFUN is a reversible case-expression employing the *symmetric first-match policy*: The control expression is matched against the patterns in the order they are given (as in, *e.g.*, the ML-family of languages), but, for the case-expression to be defined, once a match is found, any value produced by the matching branch must *not* match patterns that could have been produced by a previous branch. This policy guarantees reversibility. Perhaps surprising is the fact that recursion works in RFUN completely analogously to the way it works irreversibly; *i.e.*, using a call stack. In particular, inversion of recursive functions is handled simply by replacing the recursive call with a call to the inverse, and inverting the remainder of the function body. As such, the inverse of a recursive function is, again, a recursive function. This point will prove important later on.

An example of an RFUN program for computing Fibonacci-pairs is shown in Fig. 1 [22,25]: Given a natural number n encoded in unary, *fib*(n) produces the pair $\langle f_{n+1}, f_{n+2} \rangle$ where f_i is the unary encoding of the i'th Fibonacci number. Notice the use of the duplication operator in the definition of *plus*: The duplication-equality operator on the right-hand side of the first branch of *plus* duplicates $\langle x \rangle$ into $\langle x, x \rangle$ in the forward direction, and checks the equality of two values $\langle x, y \rangle$ in the backward direction. This accounts for the fact that the first branch of *plus* always returns two identical values, while the second branch always returns two different values. The first-match policy of RFUN described above guarantees the reversibility of the auxiliary function *plus*, which is defined by *plus* $\langle x, y \rangle = \langle x, x + y \rangle$.

Please cite this article in press as: R. Kaarsgaard et al., Join inverse categories and reversible recursion, J. Log. Algebraic Methods Program. (2016), http://dx.doi.org/10.1016/j.jlamp.2016.08.003