THE LOGIC OF REVERSIBLE COMPUTING

Theory and Practice

Robin Kaarsgaard

February 26, 2018

DIKU, Department of Computer Science, University of Copenhagen
robin@di.ku.dk
http://www.di.ku.dk/~robin

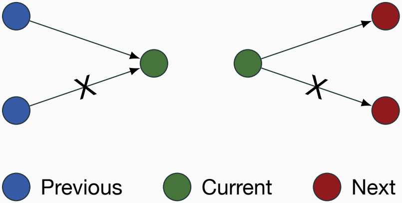
We tend to think of scientists as devices with the signature

$$\text{Funding} \otimes \text{Coffee} \xrightarrow{\textit{Scientist}} \text{Science} \otimes \text{Noise}$$

Noise: Opinions, essays titled “XYZ considered harmful”, etc.

- Reversible computing: What, how, why?
- Reversibility from a denotational perspective
- Theme: Reversible recursion
- Models of reversible programming languages
- Other work
- Concluding remarks

Reversible computing is the study of models of computation that exhibit both *forward* and *backward determinism*.



As a consequence, reversible computers are just as happy running backwards as they are running forward.

Functions computed by reversible means are *injective*.

"I'm sorry, wait… you want to make computers do what?"

Information is physical.

Landauer: Erasing information, *no matter how you do it*, costs energy: *at least* $kT\log(2)$ joules per bit of information, to be precise.

Reversible computing: Computing without information erasure – avoids Landauer limit, potential to reduce power consumption of computing machinery.

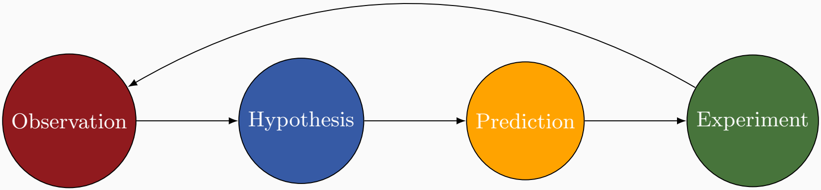
Incidental applications: Naturally invertible problems, has even seen applications in the programming of assembly robots(!)

"So what is it that you do exactly*?"*

"Caution!!!! Live bees // Part of a Master's thesis study"

```
$,='';sub f{my($a,$r)=@_;@$a-$_||print@$a;
for$c(0..$_-1){my($i,$b);for(@$a){$b=1,last
if$c==$_||abs$c-$_==$r-$i++}!$b&&f(
$A=[@$a,$c],$r+1)&&return$A}}f([])
```

(Credit: User vakorol at jagc.org)



Hypothesis formulated as a *mathematical model*, predictions extracted from this. Experiments replaced by formal proofs.

Mathematical modelling tool of choice: Category theory.

Starting point: Inverse categories – categories where each morphism $X \xrightarrow{f} Y$ has a unique *partial inverse* $Y \xrightarrow{f^\dagger} X$ such that $f \circ f^\dagger \circ f = f$ and $f^\dagger \circ f \circ f^\dagger = f^\dagger$.

Canonical example: The category **PInj** of sets and partial injective functions.

Thesis (B. G. Giles): Inverse categories are semantic domains for reversible computation.

However, partial invertibility is not enough: This is closer to injectivity than to reversibility, and we need to be able to separate the two.

B. G. Giles, *An investigation of some theoretical aspects of reversible computing*, 2014.

Idea: Exploit compositionality.

A program p is said to be reversible iff for every meaningful subprogram p' of p, $[\![p']\!]$ is partially invertible.

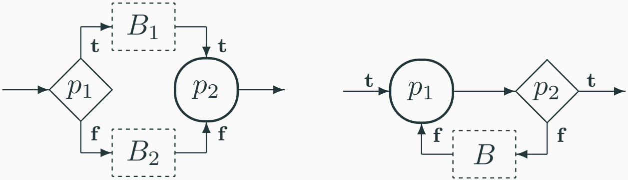
Compositionality also seems central to the operational understanding of reversibility: A program is reversible if it only performs reversible primitive operations, and if these operations are combined in a way that preserves this property.

Thesis (me): Reversible programs have compositional semantics.

When you're first taught about reversible programming, the programming language Janus is usually the starting point.

Janus looks very similar to other procedural languages; it has atomic state update commands, while loops, conditionals, etc.

However, the latter two look a little differently than usual.



Reversible while loops here perform reversible *tail recursion*.

T. Yokoyama, R. Glück, *A Reversible Programming Language and its Invertible Self-Interpreter*, 2007

Then, you graduate to Rfun: A reversible functional programming language (originally in the style of LISP/Scheme).

Save for a strange operator (duplication/equality) and some semantic conditions on case-expressions, it is virtually indistinguishable from ordinary functional programming languages in that style.

...save for the ability to *uncall* functions (*i.e.*, call the inverse function).

It even supports general recursion, which works exactly as it does irreversibly (*i.e.*, using a call stack).

T. Yokoyama, H. B. Axelsen, R. Glück, *Towards a Reversible Functional Language*, 2011

$$
\begin{aligned}
\mathcal{I}_p[\![d^*]\!] &= \mathcal{I}_d[\![d]\!]^* \\
\mathcal{I}_d[\![f\ l \triangleq e]\!] &= f^{-1}\ x \triangleq \mathbf{case}\ x\ \mathbf{of}\ \mathcal{I}[\![e, l]\!] \\
\mathcal{I}[\![l, e]\!] &= \{l \to e\} \\
\mathcal{I}[\![\mathbf{let}\ l_1 = f\ l_2\ \mathbf{in}\ e', e]\!] &= \mathcal{I}[\![e', \mathbf{let}\ l_2 = f^{-1}\ l_1\ \mathbf{in}\ e]\!] \\
\mathcal{I}[\![\mathbf{rlet}\ l_1 = f\ l_2\ \mathbf{in}\ e', e]\!] &= \mathcal{I}[\![e', \mathbf{rlet}\ l_2 = f^{-1}\ l_1\ \mathbf{in}\ e]\!] \\
\mathcal{I}[\![\mathbf{case}\ l\ \mathbf{of}\ \{p_i \to e_i\}_{i=1}^{m}, e]\!] &= \cup_{i=1}^{m}(\mathbf{if}\ \sigma_i \neq \bot\ \mathbf{then}\ \mathcal{I}[\![e_i, \sigma_i e]\!] \\
&\qquad\quad \mathbf{else}\ \mathcal{I}[\![e_i, \mathbf{case}\ p_i\ \mathbf{of}\ l \to e]\!]) \\
&\quad\ \mathbf{where}\ \sigma_i \text{ is the unification of } l \text{ and } p_i
\end{aligned}
$$

The inverse to a recursive function is a recursive function constructed by inverting the function body, replacing the (original) recursive call with a recursive call to the thus constructed inverse.

T. Yokoyama, H. B. Axelsen, R. Glück, *Towards a Reversible Functional Language*, 2011

In summary:

- Tail recursion (as in Janus) requires some surgery to work reversibly.
- General recursion (as in Rfun) just works reversibly as usual, and it even comes with nice inversion properties included.

What is going on here?!

“I don’t know… we’ve always done it that way.”

A friend in need: Domain theory.

Join inverse categories: Inverse categories equipped with an operator $\vee$ for "gluing" parallel maps together if they are somehow compatible.

Theorem: Every join inverse category is canonically enriched in the category of directed-complete partial orders and continuous maps.

As a consequence, every functional $\varphi : \mathscr{C}(X, Y) \to \mathscr{C}(X, Y)$ has a least fixed point $\mathrm{fix}\ \varphi : X \to Y \quad \Rightarrow \quad$ general recursion!

X. Guo, *Products, Joins, Meets, and Ranges in Restriction Categories*, 2012

Even better: The inverse to such fixed points may be constructed *exactly* as Rfun prescribes!

Theorem: Every functional $\varphi : \mathscr{C}(X,Y) \to \mathscr{C}(X,Y)$ has a *fixed point adjoint* $\overline{\varphi} : \mathscr{C}(Y,X) \to \mathscr{C}(Y,X)$ satisfying $(\mathrm{fix}\ \varphi)^\dagger = \mathrm{fix}\ \overline{\varphi}$.

Trick: Define $\overline{\varphi}(f) = \varphi(f^\dagger)^\dagger$ (just like the Rfun program inverter instructed).

A join-preserving *disjointness tensor*: A "sum-like" symmetric monoidal tensor $(-) \oplus (-)$ that preserves joins in each component. Specifically has injections

$$X \xrightarrow{\amalg_1} X \oplus Y \qquad Y \xrightarrow{\amalg_2} X \oplus Y$$

Such a join inverse category is also a *(strong) unique decomposition category*.

B. G. Giles, *An investigation of some theoretical aspects of reversible computing*, 2014.

E. Haghverdi, *A categorical approach to linear logic, geometry of proofs and full completeness*, 2000

N. Hoshino, *A Representation Theorem for Unique Decomposition Categories*, 2012

In particular, it has a categorical trace given by the *trace formula*

$$\mathrm{Tr}(f) = \left(\bigvee_{n \in \omega} f_{21} \circ f_{22}^{n} \circ f_{12} \right) \vee f_{11}$$

where $f_{ij} = \amalg_j^\dagger \circ f \circ \amalg_i$.

This is a *dagger trace*: It satisfies $\mathrm{Tr}(f^\dagger) = \mathrm{Tr}(f)^\dagger$.

This is *precisely* what the reversible functional programming language Theseus uses for reversible (tail) recursion. Can also be used to model reversible while loops (more on this later).

E. Haghverdi, *A categorical approach to linear logic, geometry of proofs and full completeness*, 2000

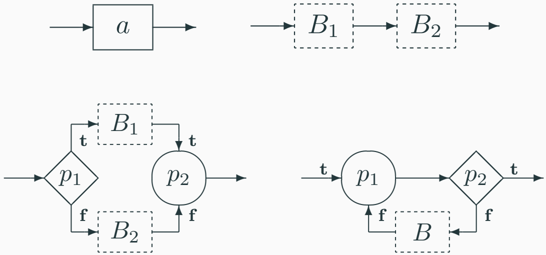
P. Selinger, *A survey of graphical languages for monoidal categories*, 2011

R. P. James, A. Sabry, *Theseus: A High Level Language for Reversible Computing*, 2014

STRUCTURED REVERSIBLE FLOWCHART LANGUAGES

Put the “abstract nonsense” to work: Denotational semantics for structured reversible flowchart languages.

Structured reversible flowchart language: A reversible imperative language with a number of *atomic steps* and *predicates* which may be combined using the following four flowchart structures.



T. Yokoyama, H. B. Axelsen, R. Glück, *Fundamentals of reversible flowchart languages*, 2015

Example: The family $\texttt{RINT}_k$. Reversible programming with k integer variables available (assumed zero-cleared at beginning).

$$
\begin{array}{llr}
p ::= & \texttt{true} \mid \texttt{false} \mid \texttt{x}_\texttt{i} = 0 & \text{(Atomic predicates)} \\
 & \mid p \;\texttt{and}\; p \mid \texttt{not}\; p & \text{(Boolean operators)} \\
c ::= & \texttt{x}_\texttt{i} \mathrel{+}= \texttt{x}_\texttt{j} \mid \texttt{x}_\texttt{i} \mathrel{-}= \texttt{x}_\texttt{j} \mid \texttt{x}_\texttt{i} \mathrel{+}= \overline{n} & \text{(Atomic steps)} \\
 & \mid c \,;\, c & \text{(Sequencing)} \\
 & \mid \texttt{if}\; p \;\texttt{then}\; c \;\texttt{else}\; c \;\texttt{fi}\; p & \text{(Conditionals)} \\
 & \mid \texttt{from}\; p \;\texttt{loop}\; c \;\texttt{until}\; p & \text{(Loops)}
\end{array}
$$

Other examples: Janus (without recursion), `R-WHILE`, `R-CORE`.

R. Glück, T. Yokoyama, *A Linear-Time Self-Interpreter of a Reversible Imperative Language*, 2016

R. Glück, T. Yokoyama, *A Minimalist's Reversible While Language*, 2017

Immediate roadblock: How do we represent Boolean predicates reversibly? (Like everything else, they may diverge on some inputs!)

Representing Boolean predicates on X as morphisms

$$X \xrightarrow{p} 1 + 1$$

doesn't work – no coproducts, terminal object degenerate.

$$X \xrightarrow{p} I \oplus I$$

for suitable distinguished object I – generally not invertible.

Better: As morphisms

$$X \xrightarrow{p} X \oplus X$$

which additionally satisfy that they only tag inputs with either left or right, but does not change them in any way.

Convention: Things sent to the left are considered *true*, things sent to the right are considered *false*.

Morphisms *very* similar to these are known in the literature as *decisions*. Adapting to inverse categories:

Extensive inverse category: An inverse category with a disjointness tensor in which each map $X \xrightarrow{f} Y \oplus Z$ has a unique *decision* $X \xrightarrow{\langle f \rangle} X \oplus X$ (axioms omitted).

R. Cockett, S. Lack, *Restriction categories III: colimits, partial limits and extensivity*, 2007

“[A decision is a map which] decides which branch to take, but doesn’t yet do any actual work”

We can do Boolean operations and constants this way as well, *e.g.*

$$\llbracket tt \rrbracket = \amalg_1$$
$$\llbracket ff \rrbracket = \amalg_2$$
$$\llbracket \mathsf{not}\ p \rrbracket = \gamma \circ \llbracket p \rrbracket$$

(Conjunction and disjunction also possible, but too gory to show in detail!)

Observation: The partial inverse to a predicate is precisely its corresponding assertion.

A join inverse category with a join-preserving disjointness tensor (specifically an extensive inverse category) equipped with

- Distinguished objects I (with some properties) and Σ such that states have an interpretation as *total* morphisms

$$[\![\sigma]\!] : I \to \Sigma \quad ,$$

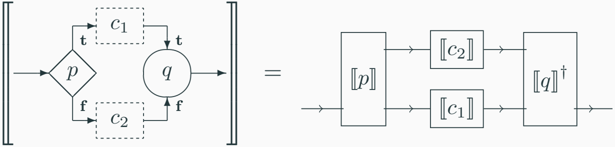
- interpretations of *atomic steps* as morphisms

$$[\![c]\!] : \Sigma \to \Sigma \quad ,$$

- and interpretations of *atomic predicates* as decisions on Σ,

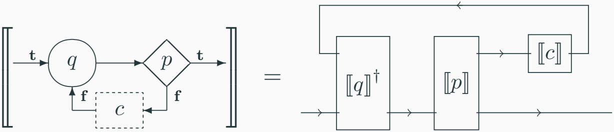
$$[\![p]\!] : \Sigma \to \Sigma \oplus \Sigma \quad .$$

- By previous slide, we may close atomic predicates under Boolean operations.



$$[\![\text{if } p \text{ then } c_1 \text{ else } c_2 \text{ fi } q]\!] = [\![q]\!]^\dagger \circ ([\![c_1]\!] \oplus [\![c_2]\!]) \circ [\![p]\!]$$

LOOPS



$$[\![\text{from } q \text{ do } c \text{ until } p]\!] = \mathrm{Tr}((\mathrm{id}_{\Sigma} \oplus [\![c]\!]) \circ [\![p]\!] \circ [\![q]\!]^{\dagger})$$

Omitting 15 dense pages of math and an operational semantics, we obtain the following correspondence theorem:

Soundness and adequacy: For any program p and state σ, $[\![p]\!] \circ [\![\sigma]\!]$ is total iff there exists σ' such that $\sigma \vdash p \downarrow \sigma'$.

- That $[\![p]\!] \circ [\![\sigma]\!]$ is total amounts to saying that p converges *denotationally* in σ.
- That there exists σ' such that $\sigma \vdash p \downarrow \sigma'$ means that p converges *operationally* in σ.

Soundness and adequacy (again): The operational and denotational notions of convergence are in agreement.

Further, when some additional conditions are met, we may even obtain full abstraction:

Full abstraction: For all commands c_1 and c_2, $c_1 \approx c_2$ iff $[\![c_1]\!] = [\![c_2]\!]$.

- $(-) \approx (-)$ is the usual *observational equivalence*: $c_1 \approx c_2$ if for all states σ, $\sigma \vdash c_1 \downarrow \sigma'$ iff $\sigma \vdash c_2 \downarrow \sigma'$ (note contextual equivalence not needed!).
- $[\![c_1]\!] = [\![c_2]\!]$ is *equality of interpretations* as morphisms in the category.

Full abstraction (again): Commands are operationally equivalent iff they are equal on their interpretations.

Full abstraction (one more time): The operational and denotational notions of command equivalence are in agreement.

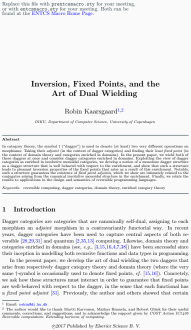
Problem: Showing correctness of program inverter doable but laborious with operational semantics. By induction on program c with hypothesis $[\![c]\!]^\dagger = [\![\mathrm{Inv}(c)]\!]$.

$$\mathrm{Inv}(\textbf{from } p \textbf{ loop } c' \textbf{ until } q) = \textbf{from } q \textbf{ loop } \mathrm{Inv}(c') \textbf{ until } p$$

We can derive this as follows:

$$\begin{aligned}
[\![\textbf{from } p \textbf{ loop } c' \textbf{ until } q]\!]^\dagger &= \mathrm{Tr}((\mathrm{id}_\Sigma \oplus [\![c']\!]) \circ [\![q]\!] \circ [\![p]\!]^\dagger)^\dagger \\
&= \mathrm{Tr}(((\mathrm{id}_\Sigma \oplus [\![c']\!]) \circ [\![q]\!] \circ [\![p]\!]^\dagger)^\dagger) \\
&= \mathrm{Tr}([\![p]\!] \circ [\![q]\!]^\dagger \circ (\mathrm{id}_\Sigma \oplus [\![c']\!]^\dagger)) \\
&= \mathrm{Tr}((\mathrm{id}_\Sigma \oplus [\![c']\!]^\dagger) \circ [\![p]\!] \circ [\![q]\!]^\dagger) \\
&= \mathrm{Tr}((\mathrm{id}_\Sigma \oplus [\![\mathrm{Inv}(c')]\!]) \circ [\![p]\!] \circ [\![q]\!]^\dagger) \\
&= [\![\textbf{from } q \textbf{ loop } \mathrm{Inv}(c') \textbf{ until } p]\!] \\
&= [\![\mathrm{Inv}(\textbf{from } p \textbf{ loop } c' \textbf{ until } q)]\!]
\end{aligned}$$

"That's all well and good, but what else have you been doing with your life?"



Replace this file with `prentcsmacro.sty` for your meeting, or with `entcsmacro.sty` for your meeting. Both can be found at the ENTCS Macro Home Page.

Inversion, Fixed Points, and the Art of Dual Wielding

Robin Kaarsgaard[1,2]

DIKU, Department of Computer Science, University of Copenhagen

Abstract

In category theory, the symbol † ("dagger") is used to denote (at least) two very different operations on morphisms: Taking their *adjoint* (in the context of dagger categories) and finding their *least fixed point* (in the context of domain theory and categories enriched in domains). In the present paper, we wield both of these daggers at once and consider dagger categories enriched in domains. Exploiting the view of dagger categories as enriched in involutive monoidal categories, we develop a notion of a monotone dagger structure as a dagger structure that is well behaved with respect to the enrichment, and show that such a structure leads to pleasant inversion properties of the fixed points that arise as a result of this enrichment. Notably, such a structure guarantees the existence of *fixed point adjoints*, which we show are intimately related to the conjugates arising from the canonical involutive monoidal structure in the enrichment. Finally, we relate the results to applications in the design and semantics of reversible programming languages.

Keywords: reversible computing, dagger categories, domain theory, enriched category theory

1 Introduction

Dagger categories are categories that are canonically self-dual, assigning to each morphism an *adjoint* morphism in a contravariantly functorial way. In recent years, dagger categories have been used to capture central aspects of both reversible [28,29,31] and quantum [2,35,13] computing. Likewise, domain theory and categories enriched in domains (see, *e.g.*, [3,15,16,4,7,38]) have been successful since their inception in modelling both recursive functions and data types in programming.

In the present paper, we develop the art of dual wielding the two daggers that arise from respectively dagger category theory and domain theory (where the very same †-symbol is occasionally used to denote fixed points, *cf.* [15,16]). Concretely, we ask how these structures must interact in order to guarantee that fixed points are well-behaved with respect to the dagger, in the sense that each functional has a *fixed point adjoint* [31]. Previously, the author and others showed that certain

[1] Email: `robin@di.ku.dk`

[2] The author would like to thank Martti Karvonen, Mathys Rennela, and Robert Glück for their useful comments, corrections, and suggestions; and to acknowledge the support given by *COST Action IC1405 Reversible computation: Extending horizons of computing.*

©2017 Published by Elsevier Science B. V.

More work on reversible recursion: Are fixed point adjoints unique to models of classical reversible computing? (No.) Are they canonical somehow? (Yes.) Is there a similar notion for parametrized fixed points? (Yes.) etc.

Ricercar: A Language for Describing and Rewriting Reversible Circuits with Ancillae and Its Permutation Semantics

Michael Kirkedal Thomsen[1](✉), Robin Kaarsgaard[2], and Mathias Soeken[1]

[1] Group of Computer Architecture, University of Bremen, Bremen, Germany
{kirkedal,msoeken}@informatik.uni-bremen.de
[2] DIKU, Department of Computer Science,
University of Copenhagen, Copenhagen, Denmark
robin@di.ku.dk

Abstract. Previously, Soeken and Thomsen presented six basic semantics-preserving rules for rewriting reversible logic circuits, defined using the well-known diagrammatic notation of Feynman. While this notation is both useful and intuitive for describing reversible circuits, its shortcomings in generality complicates the specification of more sophisticated and abstract rewriting rules.

In this paper, we introduce *Ricercar*, a general textual description language for reversible logic circuits designed explicitly to support rewriting. Taking the not gate and the identity gate as primitives, this language allows circuits to be constructed using control gates, sequential composition, and ancillae, through a notion of *ancilla scope*. We show how the above-mentioned rewriting rules are defined in this language, and extend the rewriting system with five additional rules to introduce and modify ancilla scope. This treatment of ancillae addresses the limitations of the original rewriting system in rewriting circuits with ancillae in the general case.

To set Ricercar on a theoretical foundation, we also define a permutation semantics over symmetric groups and show how the operations over permutations as transposition relate to the semantics of the language.

Keywords: Reversible logic · Term rewriting · Ancillae · Circuit equivalence · Permutation

1 Introduction

In [14] two of the authors presented six elementary rules for rewriting reversible circuits using mixed-polarity multiple-control Toffoli gates. Building on this,

M.K. Thomsen—This work was partly funded by the *European Commission* under the *7th Framework Programme*.
M.K. Thomsen—A preliminary version of Ricercar was presented as work-in-progress at *6th Conference on Reversible Computation, 2014*.

J. Krivine and J.-B. Stefani (Eds.): RC 2015, LNCS 9138, pp. 200–215, 2015.
DOI: 10.1007/978-3-319-20860-2_13

A Classical Propositional Logic for Reasoning About Reversible Logic Circuits

Holger Bock Axelsen, Robert Glück, and Robin Kaarsgaard(✉)

DIKU, Department of Computer Science,
University of Copenhagen, Copenhagen, Denmark
{funkstar,glueck,robin}@di.ku.dk

Abstract. We propose a syntactic representation of reversible logic circuits in their entirety, based on Feynman's control interpretation of Toffoli's reversible gate set. A pair of interacting proof calculi for reasoning about these circuits is presented, based on classical propositional logic and monoidal structure, and a natural order-theoretic structure is developed, demonstrated equivalent to Boolean algebras, and extended categorically to form a sound and complete semantics for this system. We show that all strong equivalences of reversible logic circuits are provable in the system, derive an equivalent equational theory, and describe its main applications in the verification of both reversible circuits and template-based reversible circuit rewriting systems.

1 Introduction

Reversible computing–the study of computing models deterministic in both the forward and backward directions–is primarily motivated by a potential to reduce the power consumption of computing processes, but has also seen applications in topics such as static average-case program analysis [17], unified descriptions of parsers and pretty-printers [16], and quantum computing [6]. The potential energy reduction was first theorized by Rolf Landauer in the early 1960s [12], and has more recently seen experimental verification [2,14]. Reaping these potential benefits in energy consumption, however, requires the use of a specialized gate set guaranteeing reversibility, when applied at the level of logic circuits.

Boolean logic circuits correspond immediately to propositions in classical propositional logic (CPL): This is done by identifying input lines with propositional atoms, and logic gates with propositional connectives, reducing the problem of reasoning about circuits to that of reasoning about arbitrary propositions in a classical setting. However, although Toffoli's gate set for reversible circuit logic is equivalent to the Boolean one in terms of what can be computed [22], it falls short of this immediate and pleasant correspondence. This article seeks

The authors acknowledge support from the Danish Council for Independent Research | Natural Sciences under the *Foundations of Reversible Computing* project, and partial support from COST Action IC1405 *Reversible Computation*.
Colors in electronic version.

J. Väänänen et al. (Eds.): WoLLIC 2016, LNCS 9803, pp. 52–67, 2016.
DOI: 10.1007/978-3-662-52921-8_4

Rewriting of reversible circuits: Two approaches to rewriting of reversible circuits. One, a programming language with an equational theory: Practical but possibly incomplete. The other, a formal logic: Considerably less practical but complete.

Reversible effects as inverse arrows

Chris Heunen[1], Robin Kaarsgaard[2], and Martti Karvonen[3]

[1] University of Edinburgh, chris.heunen@ed.ac.uk
[2] University of Copenhagen, robin@di.ku.dk
[3] University of Edinburgh, martti.karvonen@ed.ac.uk

Abstract. Reversible computing models settings in which all processes can be reversed. Applications include low-power computing, quantum computing, and robotics. It is unclear how to represent side-effects in this setting, because conventional methods need not respect reversibility. We model reversible effects by adapting Hughes' arrows to dagger arrows and inverse arrows. This captures several fundamental reversible effects, including concurrency and mutable store computations. Whereas arrows are monoids in the category of profunctors, dagger arrows are involutive monoids in the category of profunctors, and inverse arrows satisfy certain additional properties. These semantics inform the design of functional reversible programs supporting side-effects.

Keywords: Reversible Effect; Arrow; Inverse Category; Involutive Monoid

1 Introduction

Reversible computing studies settings in which all processes can be reversed: programs can be run backwards as well as forwards. Its history goes back at least as far as 1961, when Landauer formulated his physical principle that logically irreversible manipulation of information costs work. This sparked the interest in developing reversible models of computation as a means to making them more energy efficient. Reversible computing has since also found applications in high-performance computing [29], process calculi [8], probabilistic computing [32], quantum computing [31], and robotics [30].

There are various theoretical models of reversible computations. The most well-known ones are perhaps Bennett's reversible Turing machines [4] and Toffoli's reversible circuit model [33]. There are also various other models of reversible automata [26, 24] and combinator calculi [1, 20].

We are interested in models of reversibility suited to functional programming languages. Functional languages are interesting in a reversible setting for two reasons. First, they are easier to reason and prove properties about, which is a boon if we want to understand the logic behind reversible programming. Second, they are not stateful by definition, which makes it easier to reverse programs. It is fair to say that existing reversible functional programming languages [21, 34] still lack various desirable constructs familiar from the irreversible setting.

Irreversible functional programming languages like Haskell naturally take semantics in categories. The objects interpret types, and the morphisms interpret

Effects in reversible functional programming: What is a good way to structure reversible effects for reversible functional programming? Monads don't seem to work, even ones that are particularly nice. However, Arrows *can* be adjusted to play well with inversion.

RFun Revisited

Robin Kaarsgaard and Michael Kirkedal Thomsen

DIKU, Department of Computer Science, University of Copenhagen
{robin, m.kirkedal}@di.ku.dk

We describe here the steps taken in the further development of the reversible functional programming language RFun. Originally, RFun was designed as a first-order untyped language that could only manipulate constructor terms; later it was also extended with restricted support for function pointers [6, 5]. We outline some of the significant updates ot the language, including a static type system based on relevant typing, with special support for ancilla (read-only) variables added through an unrestricted fragment. This has further resulted in a complete makeover of the syntax, moving towards a more modern, Haskell-like language.

Background In the study of reversible computation, one investigates computational models in which individual computation steps can be uniquely and unambiguously inverted. For programming languages, this means languages in which programs can be run *backward* and get a unique result (the exact input).

Though the field is often motivated by a desire for energy and entropy preservation though the work of Landauer [3], we are more interested in the possibility to use reversibility as a property that can aid in the execution of a system; an approach which can be credited to Huffman [1]. In this paper we specifically consider RFun. Another notable example of a reversible functional language is Theseus [2], which has also served as a source of inspiration for some of the developments described here.

Ancillae Ancillae (considered ancillary variables in this context) is a term adopted from physics to describe a state in which entropy is unchanged. Here we specifically use it for variables for which we can guarantee that their values are unchanged over a function call. We cannot put too little emphasis on the guarantee, because we have taken a conservative approach and will only use it when we statically can ensure that it is upheld.

1 RFun version 2

In this section, we will describe the most interesting new additions to RFun and how they differ from the original work. Rather than showing the full formalisation, we will instead argue for their benefits to a reversible (functional) language.

Figure 1 shows an implementation of the Fibonacci function in RFun, which we will use as a running example. Since the Fibonacci function is not injective (the first and second Fibonacci numbers are both 1), we instead compute *Fibonacci pairs*, which *are* unique. Hence, the first Fibonacci pair is $(0, 1)$, the second to $(1, 1)$, third $(2, 1)$, and so forth.

The implementation in RFun can be found in Figure 1 and consists of a type definition `Nat` and two functions `plus` and `fib`. Here, `Nat` defines the natural numbers as Peano numbers, `plus` implements addition over the defined natural numbers, while `fib` is the implementation of the Fibonacci pairfunction. Further, Figure 2 shows an implementation of the map function.

The future of Rfun: A brief vision, including ideas for a type system supporting both ancillary and dynamic variables.

- The internal logic of extensive restriction/inverse categories.
- Decisions and reversible functional programming.

- Reversible computing – an emerging computing paradigm with physical implications.
- Reversible programming languages as seen through the lens of category theory.
- Focus: Understanding reversible recursion.

Thank you for attending!

ACKNOWLEDGEMENTS