CoreFun: A Typed Functional Reversible Core Language

Petur Andrias Højgaard Jacobsen, Robin Kaarsgaard, and Michael Kirkedal Thomsen

DIKU, Department of Computer Science, University of Copenhagen
`xqw428@alumni.ku.dk, {robin, m.kirkedal}@di.ku.dk`

Abstract. This paper presents CoreFun, a typed reversible functional language, which seeks to reduce typed reversible functional programming to its essentials. We present a complete formal definition of the language, including its formal semantics and type system, the latter of which is based on a combined reasoning logical system of unrestricted and relevantly typed terms, and allows special support for ancillary (read-only) variables through its unrestricted fragment. We show how, in many cases, the type system provides the possibility to statically check for the reversibility of programs. Finally, we detail how higher-level language features such as variants and type classes may be incorporated into CoreFun as syntactic sugar, such that CoreFun may be used as a core language for a reversible functional language in a more modern style.

Keywords: reversible computation, functional programming, programming languages, types, formal semantics

1 Introduction

Reversible computing is the study of 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). In this paper, we restrict ourselves to *garbage-free* reversible programming languages, which guarantee not only that all programs are reversible, but also that no hidden duplication of data is required in order to make this guarantee.

In this paper we present a simple, but *r-Turing complete* [2], reversible typed functional programming language, CoreFun. Functional languages and programming constructs are currently quite successful; this includes both applications in special domains, e.g. Erlang, and functional constructs introduced in mainstream programming languages, such as Java and C++. We believe that functional languages also provide a suitable environment for studying reversible programs and computations, as recently shown in [19]. However, the lack of a type system exposed the limitations of the original RFun language, which has motivated this work. A carefully designed type system can provide better handling of static information through the introduction of *ancillae typed* variables, which are guaranteed to be unchanged across function calls. Further, it can often be used to statically verify the *first match policy* that is essential to reversibility of partially defined functions. It should be noted that this type system is not meant to guarantee reversibility of well-typed programs (rather, guaranteeing reversibility is a job for the *semantics*). Instead, the type system aids in the clarity of expression for programs, provides fundamental well-behavedness guarantees, and is a source of additional static information which can enable static checking of certain properties, such as the aforementioned *first-match policy*.

An implementation of the work in this paper can be found at

https://github.com/diku-dk/coreFun/

1.1 Background

Initial studies of reversible (or information lossless) computationdateback to the years around 1960. These studies were based on quite different computation models and motivations: Huffman studied information lossless finite state machines for their applications in data transmission [7], Landauer came to study reversible logic in his quest to determine the sources of energy dissipation in a computing system [9], and Lecerf studied reversible Turing machines for their theoretical properties [10].

Although the field is often motivated by a desire for energy and entropy preservation though the work of Landauer [9], 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 [7]. It has since been used in areas like programming languagesfor quantum computation [6], parallel computing [16], and even robotics [17]. This diversity motivates studying reversible functional programming (and other paradigms) independently, such that we can get a better understanding of how to improve reversible programming in these diverse areas.

The earliest reversible programming language (to the authors' knowledge) is Janus, an imperative language invented in the 1980's, and later rediscovered [11,23] as interest in reversible computation spread. Janus (and languages deriving from it) have since been studied in detail, so that we today have a reasonably good understanding of these kinds of reversible flowchart languages [5, 22].

Reversible functional programming languages are still at an early stage of development, and today only a few proof-of-concept languages exist. This work is founded on the initial work on RFun [19, 21], while another notable example of a reversible functional language is Theseus [8], which has recently been further developed towards a language for quantum computations [15].

The type system formulated here is based on relevance logic (originally introduced in [1], see also [3]), a substructural logic similar to linear logic [4, 20] which (unlike linear logic) permits the duplication of data. In reversible functional programming, linear type systems (see e.g. [8]) have played an important role in ensuring reversibility, but they also appear in modern languages like the Rust programming language. To support ancillary variables at the type level, we adapt a type system inspired by Polakow's combined reasoning system of ordered, linear, and unrestricted intuitionistic logic [14].

The rest of this paper is organised in the following way: In Sect. 2 we will first introduce CoreFun followed by the type system and operational semantics. We also discuss type polymorphism and show that the language is indeed reversible. In Sect. 3 we will show how the type system in some cases can be used to statically verify the first match policy. In Sect. 4 we show how syntactic sugar can be used to design a more modern style functional language from CoreFun. Finally in Sect. 5 we conclude.

2 Formalisation of CoreFun

The following section will present the formalisation of CoreFun. The language is intended to be minimal, but it will still accommodate future extensions to a modern style functional language. We first present a core language syntax, which will work as the base of all formal analysis. Subsequently we present typing rules and operational semantics over this language. The following is build on knowledge about implementation of type systems as explained in [12].

2.1 Grammar

A program is a collection of zero or more function definitions. Each definition must be defined over some number of input variables as constant functions are not interesting is a reversible setting. All function definitions will in interpretation be available though a static context. A typing of a program is synonymous with a typing of each function. A function is identified by a name f and takes 0 or more type parameters, and 1 or more formal parameters as inputs. Each formal parameter x is associated with a typing term τ at the time of definition for each function, which may be one of the type variables given as type parameter. The grammar is given in Fig. 1.

2.2 Type system

Linear logic is the foundation for linear type theory. In linear logic, each hypothesis must be used exactly once. Likewise, values which belong to a linear type must be used exactly once, and may not be duplicated nor destroyed. However, if we accept that functions may be partial (a necessity for *r-Turing completeness* [2]), first-order data may be duplicated reversibly. For this reason, we may relax the linearity constraint to relevance, that is that all available variables *must* be used at least once.

A useful concept in reversible programming is access to ancillae, i.e. values that remain unchanged across function calls. Such values are often used as a means to guarantee reversibility in a straightforward manner. To support such ancillary variables at the type level, a type system inspired by Polakows combined

$q ::= d^*$	Program definition
$d ::= f\ \alpha^*\ v^+ = e$	Function definition
$e ::= x$	Variable name
$\mid ()$	Unit term
$\mid \textbf{inl}(e)$	Left of sum term
$\mid \textbf{inr}(e)$	Right of sum term
$\mid (e, e)$	Product term
$\mid \textbf{let}\ l = e\ \textbf{in}\ e$	Let-in expression
$\mid \textbf{case}\ e\ \textbf{of}\ \textbf{inl}(x) \Rightarrow e, \textbf{inr}(y) \Rightarrow e$	Case-of expression
$\mid f\ \alpha^*\ e^+$	Function application
$\mid \textbf{roll}\ [\tau]\ e$	Recursive-type construction
$\mid \textbf{unroll}\ [\tau]\ e$	Recursive-type deconstruction
$l ::= x$	Definition of variable
$\mid (x, x)$	Definition of product
$v ::= x : \tau_a$	Variable declaration

Fig. 1: Grammar of CoreFun. Program variables are denoted by x, and type variables by α.

$$\begin{aligned} \tau_f &::= \tau_f \to \tau_f' \mid \tau \to \tau_f' \mid \tau \leftrightarrow \tau' \mid \forall X.\tau_f \\ \tau &::= 1 \mid \tau \times \tau' \mid \tau + \tau' \mid X \mid \mu X.\tau \\ \tau_a &::= \tau \mid \tau \leftrightarrow \tau' \end{aligned}$$

Fig. 2: Typing terms. Note that X in this figure denotes any type variable.

reasoning system of ordered, linear, and unrestricted intuitionistic logic [14] is used. The type system splits the typing contexts into two parts: a static one (containing ancillary variables and other static parts of the environment), and a dynamic one (containing variables not considered ancillary). This gives a typing judgment of $\Sigma; \Gamma \vdash e : \tau$, where Σ is the static context and Γ is the dynamic context.

We discern between two sets of typing terms: primitive types and arrow types. This is motivated by a need to be careful about how we allow manipulation of functions, as we will treat all functions as statically known.

The grammar for typing terms can be seen in Fig. 2: τ_f denotes arrow types, τ primitive types, and τ_a ancillary types (i.e., types of data that may be given as ancillary data).

Arrow types are types assigned to functions. For arrow types, we discern between primitive types and arrow types in the right component of unidirectional application. We only allow primitive types in bidirectional application. This is to restrict ancillary parameters to be bound to functions, resulting in a second-order language. It is ill-formed for an type bound in the dynamic context to be of an arrow type — in this case we would be defining a higher-order language, where functions may return new functions, which would break our assumption that all functions are statically known.

Primitive types are types assigned to expressions which evaluate to canonical values by the big step semantics. These are distinctly standard, containing sum types and product types, as well as (rank-1) parametric polymorphic types[1] and a fix point operator for recursive data types (see [13] for an introduction to the latter two concepts).

Throughout this paper, we will write $\tau_1 + \cdots + \tau_n$ for the nested sum type $\tau_1 + (\tau_2 + (\cdots + (\tau_{n-1} + \tau_n) \cdots))$ and equivalently for product types $\tau_1 \times \cdots \times \tau_n$. Similarly, as is usual, we will let arrows associate to the right.

Type rules for expressions. The typing rules for expressions are shown in Fig. 3. A combination of two features of the typing rules enforces relevant typing: (1) the restriction on the contents of the dynamic context during certain typing rules, and (2) the union operator on dynamic contexts in any rule with more than one premise.

The rules for application are split into three different rules, corresponding to application of dynamic parameters (T-App1), application of static parameters (T-App2), and type instantiation for polymorphic functions (T-PApp). Notice further the somewhat odd T-Unit-Elm rule. Since relevant type systems treat variables as resources that must be consumed, specific rules are required when data *can* safely be discarded (such as the case for data of unit type). What this rule essentially states is that *any* expression of unit type can be safely discarded; this is dual to the T-Unit rule, which states that the unique value () of unit type can be freely produced (i.e., in the empty context).

Variable Typing Restriction When applying certain axiomatic type rules (T-Var1 and T-Unit), we require the dynamic context to be empty. This is necessary to inhibit unused parts of the dynamic context from being accidentally "spilled" through the use of these rules. Simultaneously, we require that when we do use a variable from the dynamic context, the dynamic context contains exactly this variable and nothing else. This requirement prohibits the same sort of spilling.

Dynamic Context Union The union of the dynamic context is a method for splitting up the dynamic context into named parts, which can then be used separately in the premises of the rule. In logical derivations, splitting the known hypotheses is usually written as $\Gamma, \Gamma' \vdash \ldots$, but we deliberately introduce a union operator to signify that we allow an overlap in the splitting of the hypotheses. Were we not to allow overlapping, typing would indeed be linear. For example, a possible split is:

[1] A rank-1 polymorphic system may not instantiate type variables with polymorphic types.

$\boxed{\text{Judgement: } \Sigma; \Gamma \vdash e : \tau}$

$$\text{T-Var1: } \frac{}{\Sigma; \emptyset \vdash x : \tau} \Sigma(x) = \tau \qquad \text{T-Var2: } \frac{}{\Sigma; (x \mapsto \tau) \vdash x : \tau}$$

$$\text{T-Unit: } \frac{}{\Sigma; \emptyset \vdash () : 1} \qquad \text{T-Unit-Elm: } \frac{\Sigma; \Gamma \vdash e : 1 \qquad \Sigma; \Gamma' \vdash e' : \tau}{\Sigma; \Gamma \cup \Gamma' \vdash e' : \tau}$$

$$\text{T-Inl: } \frac{\Sigma; \Gamma \vdash e : \tau}{\Sigma; \Gamma \vdash \mathbf{inl}(e) : \tau + \tau'} \qquad \text{T-Inr: } \frac{\Sigma; \Gamma \vdash e : \tau}{\Sigma; \Gamma \vdash \mathbf{inr}(e) : \tau' + \tau}$$

$$\text{T-Prod: } \frac{\Sigma; \Gamma \vdash e_1 : \tau \qquad \Sigma; \Gamma' \vdash e_2 : \tau'}{\Sigma; \Gamma \cup \Gamma' \vdash (e_1, e_2) : \tau \times \tau'}$$

$$\text{T-App1: } \frac{\Sigma; \Gamma \vdash f : \tau \leftrightarrow \tau' \qquad \Sigma; \Gamma' \vdash e : \tau}{\Sigma; \Gamma \cup \Gamma' \vdash f\ e : \tau'}$$

$$\text{T-App2: } \frac{\Sigma; \Gamma \vdash f : \tau_a \rightarrow \tau_f \qquad \Sigma; \Gamma' \vdash e : \tau_a}{\Sigma; \Gamma \cup \Gamma' \vdash f\ e : \tau_f}$$

$$\text{T-PApp:} \frac{\Sigma; \Gamma \vdash f : \forall \alpha . \tau_f}{\Sigma; \Gamma \vdash f\ \tau_a : \tau_f[\tau_a/\alpha]} \qquad \text{T-Let1: } \frac{\Sigma; \Gamma \vdash e_1 : \tau' \qquad \Sigma; \Gamma', x : \tau' \vdash e_2 : \tau}{\Sigma; \Gamma \cup \Gamma' \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\text{T-Let2: } \frac{\Sigma; \Gamma \vdash e_1 : \tau' \times \tau'' \qquad \Sigma; \Gamma', x : \tau', y : \tau'' \vdash e_2 : \tau}{\Sigma; \Gamma \cup \Gamma' \vdash \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\text{T-Sum: } \frac{\Sigma; \Gamma \vdash e_1 : \tau' + \tau'' \qquad \Sigma; \Gamma, x : \tau' \vdash e_2 : \tau \qquad \Sigma; \Gamma, y : \tau'' \vdash e_3 : \tau}{\Sigma; \Gamma \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3 : \tau}$$

$$\text{T-Roll: } \frac{\Sigma; \Gamma \vdash e : \tau'[\mu X . \tau'/X]}{\Sigma; \Gamma \vdash \mathbf{roll}\ [\mu X . \tau']\ e : \mu X . \tau'} \qquad \text{T-Unroll: } \frac{\Sigma; \Gamma \vdash e : \mu X . \tau}{\Sigma; \Gamma \vdash \mathbf{unroll}\ [\mu X . \tau]\ e : \tau'[\mu X . \tau/X]}$$

Fig. 3: Expression typing.

$$\frac{\begin{array}{c}\vdots \\ \emptyset; x \mapsto 1, y \mapsto 1 \vdash \ldots\end{array} \qquad \begin{array}{c}\vdots \\ \emptyset; y \mapsto 1, z \mapsto 1 \vdash \ldots\end{array}}{\emptyset; x \mapsto 1, y \mapsto 1, z \mapsto 1 \vdash \ldots}$$

Here y is part of the dynamic context in both premises.

Type rules for function declarations. The typing rules for function declarations are shown in Fig. 4. Here T-PFun generalizes the type arguments, next T-Fun1 consumes the ancillary variables, and finally T-Fun2 handles the last dynamic variable but applying the expression typing.

We implicitly assume that pointers to previously defined functions are placed in the static context Σ as an initial step. For example, when typing an expression e in a program where a function $f\ x = e$ is defined, and we have been able to

$\boxed{\text{Judgement: } \Sigma \Vdash d : \tau}$

$$\text{T-Fun1: } \frac{\Sigma, x : \tau_a \Vdash f\ v^+ = e : \tau_f}{\Sigma \Vdash f\ x{:}\tau_a\ v^+ = e : \tau_a \to \tau_f} \qquad \text{T-Fun2: } \frac{\Sigma; (x \mapsto \tau) \vdash e : \tau'}{\Sigma \Vdash f\ x{:}\tau = e : \tau \leftrightarrow \tau'}$$

$$\text{T-PFun: } \frac{\Sigma \Vdash f\ \beta^*\ v^+ = e : \tau_f}{\Sigma \Vdash f\ \alpha\ \beta^*\ v^+ = e : \forall \alpha.\tau_f}\alpha \notin \mathrm{TV}(\Sigma)$$

Fig. 4: Function typing.

establish that $\Sigma \Vdash f\ x = e : \tau \leftrightarrow \tau'$ for some types τ, τ', we assume that a variable $f : \tau \leftrightarrow \tau'$ is placed in the static context in which we will type e ahead of time. This initial step amounts to a typing rule for the full program.

Note that we write two very similar application rules T-App1 and T-App2. This discerns between function application of ancillary and dynamic data, corresponding to the two different arrow types. In particular, as shown in T-App1, application in the dynamic variable of a function is only possible when that function is of type $\tau \leftrightarrow \tau'$, where τ and τ' are non-arrow types: This specifically disallows higher-order functions.

2.3 Recursive and polymorphic types

The type theory of CoreFun supports both recursive types as well as rank-1 parametrically polymorphic types. To support both of these, type variables, which serve as holes that may be plugged by other types, are used.

For recursive types, we employ a standard treatment of iso-recursive types in which explicit **roll** and **unroll** constructs are added to witness the isomorphism between $\mu X.\tau$ and $\tau[\mu X.\tau/X]$ for a given type τ (which, naturally, may contain the type variable X). For a type τ, we let $\mathrm{TV}(\tau)$ denote the set of type variables that appear free in τ. For example, the type of lists of a given type τ can be expressed as the recursive type $\mu X.1 + (\tau \times X)$, and $\mathrm{TV}(1 + (\tau \times X)) = \{X\}$ when the type τ contains no free type variables. We define TV on contexts as $\mathrm{TV}(\Sigma) = \{v \in \mathrm{TV}(\tau) \mid x : \tau \in \Sigma\}$.

For polymorphism, we use an approach similar to System F, restricted to rank-1 polymorphism. In a polymorphic type system with rank-1 polymorphism, type variables themselves cannot be instantiated with polymorphic types, but must be instantiated with concrete types instead. While this approach is significantly more restrictive than the full polymorphism of System F, it is expressive enough that many practical polymorphic functions may be expressed (e.g. ML and Haskell both employ a form of rank-1 polymorphism), while being simple enough that type inference is often both decidable and feasible in practice.

$$c ::= () \mid \mathbf{inl}(c) \mid \mathbf{inr}(c) \mid (c_1, c_2) \mid \mathbf{roll}\ [\tau]\ c$$

Fig. 5: Canonical forms.

2.4 Operational Semantics

We present a call-by-value big step operational semantics on expressions in Fig. 6, with canonical forms shown in Fig. 5. As is customary with functional languages, we use substitution (defined as usual by structural induction on expressions) to associate free variables with values (canonical forms). Since the language does not allow for values of function type, we instead use an environment p of function definitions in order to perform computations in a context (such as a program) with previously defined functions.

$\boxed{\text{Judgement: } p \vdash e \downarrow c}$

$$\text{E-Unit: } \frac{}{p \vdash () \downarrow ()} \qquad \text{E-Inl: } \frac{p \vdash e \downarrow c}{p \vdash \mathbf{inl}(e) \downarrow \mathbf{inl}(c)} \qquad \text{E-Inr: } \frac{p \vdash e \downarrow c}{p \vdash \mathbf{inr}(e) \downarrow \mathbf{inr}(c)}$$

$$\text{E-Roll: } \frac{p \vdash e \downarrow c}{p \vdash \mathbf{roll}\ [\tau]\ e \downarrow \mathbf{roll}\ [\tau]\ c} \qquad \text{E-Unroll: } \frac{p \vdash e \downarrow \mathbf{roll}\ [\tau]\ c}{p \vdash \mathbf{unroll}\ [\tau]\ e \downarrow c}$$

$$\text{E-Prod: } \frac{p \vdash e_1 \downarrow c_1 \qquad p \vdash e_2 \downarrow c_2}{p \vdash (e_1, e_2) \downarrow (c_1, c_2)} \qquad \text{E-Let: } \frac{p \vdash e_1 \downarrow c_1 \qquad p \vdash e_2[c_1/x] \downarrow c}{p \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \downarrow c}$$

$$\text{E-LetP: } \frac{p \vdash e_1 \downarrow (c_1, c_2) \qquad p \vdash e_2[c_1/x, c_2/y] \downarrow c}{p \vdash \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 \downarrow c}$$

$$\text{E-CaseL: } \frac{p \vdash e_1 \downarrow \mathbf{inl}(c_1) \qquad p \vdash e_2[c_1/x] \downarrow c}{p \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3 \downarrow c}$$

$$\text{E-CaseR: } \frac{p \vdash e_1 \downarrow \mathbf{inr}(c_1) \qquad p \vdash e_3[c_1/y] \downarrow c}{p \vdash \mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3 \downarrow c} c \notin \text{PLVal}(e_2)$$

$$\text{E-App: } \frac{p \vdash e_1 \downarrow c_1 \cdots p \vdash e_n \downarrow c_n \qquad p \vdash e[c_1/x_1,\ \cdots,\ c_n/x_n] \downarrow c}{p \vdash f\ \alpha_1 \cdots \alpha_m\ e_1\ \cdots\ e_n \downarrow c} p(f) = f\ \alpha_1 \cdots \alpha_m\ x_1 \cdots x_n = e$$

Fig. 6: Big step semantics of CoreFun.

A common problem in reversible programming is to ensure that branching of programs is done in such a way as to uniquely determine in the backward direction which branch was taken in the forward direction. Since case-expressions allow for such branching, we will need to define some rather complicated machinery of *leaf expressions*, *possible leaf values*, and *leaves* (the latter is similar to what is also used in [21]) in order to give their semantics.

$$\begin{aligned}
\text{leaves}(()) &= \{()\} \\
\text{leaves}((e_1, e_2)) &= \{(e_1', e_2') \mid e_1' \in \text{leaves}(e_1), \\
&\qquad\qquad e_2' \in \text{leaves}(e_2)\} \\
\text{leaves}(\mathbf{inl}(e)) &= \left\{\mathbf{inl}(e') \mid e' \in \text{leaves}(e)\right\} \\
\text{leaves}(\mathbf{inr}(e)) &= \left\{\mathbf{inr}(e') \mid e' \in \text{leaves}(e)\right\} \\
\text{leaves}(\mathbf{roll}\ [\tau]\ e) &= \left\{\mathbf{roll}\ [\tau]\ e' \mid e' \in \text{leaves}(e)\right\} \\
\text{leaves}(\mathbf{let}\ l = e_1\ \mathbf{in}\ e_2) &= \text{leaves}(e_2) \\
\text{leaves}(\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3) &= \text{leaves}(e_2) \cup \text{leaves}(e_3) \\
\text{leaves}(x) &= \{x\} \\
\text{leaves}(\mathbf{unroll}\ [\tau]\ e) &= \{\mathbf{unroll}\ [\tau]\ e' \mid e' \in \text{leaves}(e)\} \\
\text{leaves}(f\ e_1\ \ldots\ e_n) &= \{f\ e_1'\ \ldots\ e_n' \mid e_i' \in \text{leaves}(e_i)\}
\end{aligned}$$

Fig. 7: Definition function that computes the leaves of a program.

Say that an expression e is a *leaf expression* if it does not contain any subexpression (including itself) of the form $\mathbf{let}\ l = e_1\ \mathbf{in}\ e_2$ or $\mathbf{case}\ e_1\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e_2, \mathbf{inr}(y) \Rightarrow e_3$; the collection of leaf expressions form a set, LExpr. As the name suggests, a leaf expression is an expression that can be considered as a *leaf* of another expression. The set of leaves of an expression e, denoted leaves(e), is defined in Fig. 7.

The leaves of an expression are, in a sense, an abstract over-approximation of its possible values, save for the fact that leaves may be leaf expressions rather than mere canonical forms. We make this somewhat more concrete with the definition of the *possible leaf values* of an expression e, defined as

$$\text{PLVal}(e) = \{e' \in \text{LExpr} \mid e'' \in \text{leaves}(e), e' \rhd e''\} \tag{1}$$

where the relation $- \rhd -$ on leaf expressions is defined inductively as (the symmetric closure of)

$$\begin{aligned}
() &\rhd () \\
(e_1, e_2) &\rhd (e_1', e_2') && \text{if}\quad e_1 \rhd e_1' \text{ and } e_2 \rhd e_2' \\
\mathbf{inl}(e) &\rhd \mathbf{inl}(e') && \text{if}\quad e \rhd e' \\
\mathbf{inr}(e) &\rhd \mathbf{inr}(e') && \text{if}\quad e \rhd e' \\
\mathbf{roll}\ [\tau]\ e &\rhd \mathbf{roll}\ [\tau]\ e' && \text{if}\quad e \rhd e' \\
e &\rhd x \\
e &\rhd f\ e_1\ \ldots\ e_n \\
e &\rhd \mathbf{unroll}\ [\tau]\ e'
\end{aligned}$$

As such, the set PLVal(e) is the set of leaf expressions that can be unified, in a certain sense, with a leaf of e. Since variables, function applications, and unrolls do nothing to describe the syntactic form of possible results, we define that these may be unified with *any* expression. As such, using PLVal(e) is somewhat conservative in that it may reject definitions that are in fact reversible. Note also that PLVal(e) specifically includes all canonical forms that could be produced by e, since all canonical forms are leaf expressions as well.

In this way, if we can ensure that a canonical form c produced by a branch in a case-expression could not possibly have been produced by a *previous* branch in the case-expression, we know, in the backward direction, that c must have been produced by the current branch. This is precisely the reason for the side condition of $c \notin$ PLVal(e_2) on E-CaseR, as this conservatively guarantees that c could not have been produced by the previous branch.

It should be noted that for iterative functions this may add a multiplicative execution overhead that is equal to the size of the data structure. It was previously shown in [18], where a `plus` over Peano numbers which was linear recursive over an input number, actually had quadratic runtime. There seems to be a relation between this and normal functional programs implemented in a non-tail-recursive fashion. However, details on this must be left for future work.

It is immediate that should it not hold for an arbitrary expression e', no derivation is possible by the side condition, and the expression does not evaluate to a value. It is thus possible for a function to only be defined for certain elements of the domains of some its parameters. Later, we will look at exactly when we can statically guarantee that the side condition will hold for every possible value of the domains of the parameters.

We capture the conservative correctness of our definition of PLVal(e) with respect to the operational semantics – i.e., the property that any canonical form c arising from the evaluation of an expression e will also be "predicted" by PLVal in the sense that $c \in$ PLVal(e) – in the following theorem:

Theorem 1. *If $p \vdash e \downarrow c$ then $c \in$ PLVal(e).*

Proof. By induction on the structure of the derivation of $p \vdash e \downarrow c$.

The proof is mostly straightforward: The case for E-Unit follows trivially, as do the cases for E-Unroll and E-App since leaves of **unroll** $[\tau]$ e (respectively $f\ e_1\ \cdots\ e_n$) are all of the form **unroll** $[\tau]$ e' (respectively $f\ e'_1\ \cdots\ e'_n$), and since $e'' \rhd$ **unroll** $[\tau]$ e' (respectively $e'' \rhd f\ e'_1\ \cdots\ e'_n$) for *any* choice of e'', it follows that PLVal(**unroll** $[\tau]$ e') = PLVal($f\ e_1\ \cdots\ e_n$) = LExpr. The cases for E-Inl, E-Inr, E-Roll, and E-Prod all follow straightforwardly by induction, noting that PLVal(**inl**(e)) = $\{$**inl**$(e') \mid e' \in$ PLVal(e)$\}$, and similarly for **inr**(e), (e_1, e_2), and **roll** $[\tau]$ e. This leaves only the cases for **let** and **case** expressions, which follow using the following lemma:

Lemma 1. *For any expression e, variables $x_1, \ldots, x_n$, and canonical forms $c_1, \ldots, c_n$, PLVal($e[c_1/x_1, \ldots, c_n/x_n]$) $\subseteq$ PLVal(e).*

This lemma follows straightforwardly by structural induction on e, noting that it suffices to consider the case where e is open with free variables $x_1, \ldots, x_n$, as it holds trivially when e is closed (or when its free variables are disjoint from $x_1, \ldots, x_n$). With this lemma, showing the case for, e.g., E-Let is straightforward since $c \in \text{PLVal}(e_2[c_1/x])$ by induction, and since $\text{PLVal}(e_2[c_1/x]) \subseteq \text{PLVal}(e_2)$ by this lemma, so $c \in \text{PLVal}(e_2) = \text{PLVal}(\textbf{let } x = e_1 \textbf{ in } e_2)$ by $\text{leaves}(\textbf{let } x = e_1 \textbf{ in } e_2) = \text{leaves}(e_2)$.

2.5 Reversibility

Showing that the operational semantics are reversible amounts to showing that they exhibit both forward and backward determinism. Showing forward determinism is standard for any programming language (and holds straightforwardly in CoreFun as well), but backward determinism is unique to reversible programming languages. Before we proceed, we recall the usual terminology of *open* and *closed* expressions: Say that an expression e is closed if it contains no free (unbound) variables, and open otherwise.

Unlike imperative languages, where backward determinism is straightforwardly expressed as a property of the reduction relation $\sigma \vdash c \downarrow \sigma'$, backward determinism is somewhat more difficult to express for functional languages, as the obvious analogue – that is, if $e \downarrow c$ and $e' \downarrow c$ then $e = e'$ – is much too restrictive (specifically, it is obviously *not* satisfied in all but the most trivial reversible functional languages). A more suitable notion turns out to be a *contextual* one, where rather than considering the reduction behaviour of closed expressions in themselves, we consider the reduction behaviour of canonical forms in a given *context* (in the form of an open expression) instead.

Theorem 2 (Contextual backward determinism). *For all open expressions e with free variables $x_1, \ldots, x_n$, and all canonical forms $v_1, \ldots, v_n$ and $w_1, \ldots, w_n$, if $p \vdash e[v_1/x_1, \ldots, v_n/x_n] \downarrow c$ and $p \vdash e[w_1/x_1, \ldots, w_n/x_n] \downarrow c$ then $v_i = w_i$ for all $1 \leq i \leq n$.*

The proof of this theorem follows by induction on the structure of e. The only interesting case is for case-expressions, where the side condition of the E-CaseR rule has to be applied. We notice that injectivity of functions follows as a pleasant corollary:

Corollary 1 (Injectivity). *For all functions f and canonical forms v, w, if $p \vdash f\ v \downarrow c$ and $p \vdash f\ w \downarrow c$ then $v = w$.*

Proof. Let e be the open expression $f\ x$ (with free variable x). Since $(f\ x)[v/x] = f\ v$ and $(f\ x)[w/x] = f\ w$, applying Theorem 2 on e yields precisely that if $p \vdash f\ v \downarrow c$ and $p \vdash f\ w \downarrow c$ then $v = w$. $\square$

3 Statically checking the first match policy

The first match policy is essential when ensuring reversibility of partial functions. It is, unfortunately, a property that can only be fully guaranteed at run-time;

from Rice's theorem we know that all non-trivial semantic properties of programs are undecidable. However, with the type system, we can now in many cases resolve the first match policy statically.

For normal programs, we differentiate between two notions of divergence: (1) A function may have inputs that do not result in program termination. (2) A function may have inputs for which it does not have a defined behaviour; this could be the result of missing clauses.

Note that the semantics of CoreFun dictate that if a computation diverges in the forward direction, no backward computation can result in this specific input. Similarly for backwards computations. If the program diverges in the forward direction and not in the backward direction, we should be able to find some input to the inverse function which results in the diverging input in the forward direction. Since the inverse direction converges, we have determined a result in the forward direction, which is a contradiction.

Non-termination is not the property we will address here, but rather inputs for which the function is not defined. Because the first match policy is enforced by the operational semantics, it follows that whenever an expression does not uphold the first match policy, it cannot be derived. Thus, the domain of a function might not be the complete underlying set of its types, because some element in an underlying type may make the first match policy fail. We wish to investigate exactly when can or when we cannot *guarantee* that a function is going to uphold the first match policy for all elements of the types of its parameters. In some ways this is reminiscent of arguing for totality of a function in mathematics. This property of totality is not symmetric: More specifically, a function f and the inverse function f^{-1} might not both be total on their respective domains although it is certainly possible.

The type system can aid us in the endeavour of guaranteeing the first match policy. It formally hints us at the underlying sets of values which occur in case-expressions.

3.1 First match over open terms

Intuitively, when the range of a function call is well-defined (typed), and all the leaves are disjoint, it is clear that any evaluated term will not match any other leaf. For example, the following function performs a transformation on a sum term, and all leaves are disjoint; either **inl**$(\cdot)$ or **inr**$(\cdot)$.

```
f x : 1 + τ = case x of
                inl(()) ⇒ inr(())
                inr(y) ⇒ inl(y)
```

In Sect. 2.4 when we defined the operational semantics (cf. Fig. 6), the first match policy was given in the case-of expression by checking that the closed term of the evaluation of the second branch (**inr**$(\cdot)$) could not be a possible leaf value of the first branch (**inl**$(\cdot)$). However, the above example includes an open term that is defined over y.

Given the previous definition of PLVal(·) (Definition 1), this is actually easy to extend. PLVal(·) has already been defined to take any term (both open and closed terms). Thus, all we have to ensure is that all leaves of the second branch do not have a possible value in the first branch.

We can also apply this the other way. Assuming that the programmer intends for the function to be totally defined, we can also check if a function can fail the first match policy. For example, the following program that collapses a sum term is partially reversible when the intended domain does not include **inr**(**inl**()).

```
f x : 1 + (1 + 1) = case x of
                      inl(()) ⇒ inr(inl()))
                      inr(y) ⇒ inr(y)
```

In this case the second leaf `inr(inl()))` will be a member of the possible values of `inr`(y).

3.2 Inductive cases

The above analysis is specifically possible because we only investigate the domain of the programs, but it also makes it very conservative. Parameters of a recursive type require a more thorough analysis. Here we adhere to an *inductive principle*, which we have to define clearly. We define a `plus` function to introduce the subject:

```
succ n = roll [μX.1 + X] inr(n)

plus n₀: μX.1 + X n₁: μX.1 + X =
  case unroll [μX.1 + X] n₁ of
    inl() ⇒ (n₀, n₀)
    inr(n') ⇒ let (n'₀, n'₁) = plus n₀ n'
              in (n'₀, succ n'₁)
```

As in the well-known structural or mathematical induction, we must identify a base case for the induction hypothesis. A simple solution is to define these as the branches in which a function call to the function which is being evaluated does not occur. There might be multiple such branches without issue. Note that this does not work well with mutually recursive functions. For `plus` there is only one base case, and this is the left arm of the case-expression.

Analogously the inductive step is defined on each branch which contains a recursive call. For each recursive call the induction hypothesis says that, granted the arguments given to the recursive call, eventually one of the base cases will be hit. This is because any instance of the recursive type can only be finitely often folded, giving a guarantee of the finiteness of the decreasing chain. Though there is a catch which should be addressed: inductive proofs are only valid for *strictly decreasing* chains of elements to ensure that the recursion actually halts. For example, for `plus` we need to make sure that $n' \prec n_1$. Should the chain not be strictly decreasing, we have that the evaluation is non-terminating and the function is not defined for this input.

To tie it all together we need to show that the recursive call in the right arm of the `plus` function does indeed result in the base case in the left arm, allowing us to use the induction hypothesis to conclude that $n'_0 = n'_1$. If we are able to, we may directly treat the return value of the recursive function call as an instance of the value which the base case returns. We then continue evaluating the body in the inductive step. For `plus` we say that:

```
... ⇒ let (n₀, n₀) = plus n₀ n′
      in (n₀, succ n₀)
```

And now we can see that the case-arms are provably disjoint, giving us a static guarantee of the first match policy. However, implementing this is very complex and sometimes requires human guidance. This has therefore been left for future work.

4 Programming in CoreFun

Although CoreFun is a full r-Turing complete language, it lacks many of the convenient features of most modern functional languages. Luckily, we can encode many language constructs directly as syntactic transformations from a less notationally heavy language to the formal core language.

The procedure entails that for each piece of syntactic abstraction we can show that there is a systematic translation from the notationally light language to the core language. This allows us to introduce a number of practical improvements without the necessity to show their semantics formally past a translation scheme.

4.1 Variants

Variants are named alternatives; they generalize sum types and case-expressions. A variant is of the form:

$$\mathtt{V} = \mathtt{v}_1 \mid \cdots \mid \mathtt{v}_n$$

Constructing a variant value entails choosing exactly one of the possibilities $\mathtt{v}_1, \ldots, \mathtt{v}_n$ as a value. Then, given a variable of a variant type, we match over its possible forms to unpack the value.

We have seen that we generalize binary sums to n-ary sums by repeated sum types in the right component of $\tau_1 + \tau_2$, and that we can chain together case expressions to match the correct arm of such a sum. We choose an encoding of variants which exploits this pattern. This works because the variant constructors are ordered and will match with the respective position in the n-ary sum. For a variant which carries no data, the translation corresponds to stripping away the variant names, leaving us with the underlying sum type of all unit types:

$$\mathtt{V} = \mathtt{v}_1 \mid \cdots \mid \mathtt{v}_n \Rightarrow 1 + \cdots + 1$$

We can further extend variants to carry data by adding parameters. We allow generic type parameters by adding a type parameter to the variant declaration.

The syntax for variants becomes:

$$\texttt{V}\ \alpha^* = \texttt{v}_1\ [\tau\alpha]^* \mid \cdots \mid \texttt{v}_n\ [\tau\alpha]^*$$

Where $[\tau\alpha]^*$ signifies zero or more constructor parameters of some type (including inner variants). If exactly one parameter p is present for a constructor $\texttt{v}_i$ the type at position i in the n-ary sum is changed from the unit type to the type p.

$$\texttt{V}\ \alpha = \texttt{v}_1 \mid \texttt{v}_2\ \tau \mid \texttt{v}_3\ \alpha \Rightarrow 1 + \tau + \alpha$$

Notice that we may generalizes any parameter-less variant constructor to one with a single parameter of type unit, which we omit from the syntax.

If a variant constructor $\texttt{v}_i$ has $m \geq 2$ parameters $p_1, \ldots, p_m$, the type in the position of i in the n-ary sum is changed into a product type $p_1 \times \cdots \times p_m$.

$$\texttt{V} = \texttt{v}_1 \mid \cdots \mid \texttt{v}_i\ \tau_1 \ldots \tau_m \mid \cdots \Rightarrow 1 + \cdots + \tau_1 \times \cdots \times \tau_m + \ldots$$

There is one more common case we need to take into consideration: If any of the variant constructors for a variant `V` have a self-referencing parameter (a parameter of type `V`), the translated type of `V` is recursive and a fresh variable is designated as the recursion parameter.

$$\texttt{V} = \texttt{v}_1 \mid \texttt{v}_2\ \texttt{V} \Rightarrow \mu X.1 + X$$

The above actually corresponds to an encoding of the natural numbers.

When the variant declarations have been translated, the occurrences of each variant type are substituted with the respective translation and expressions of variant constructors are translated into a nested structure of **Inr** and **Inl** depending on position of the variant constructor in the n-ary sum type.

A handy result to keep in mind is that if two variant definitions have the same number of alternatives, they are isomorphic and may be encoded the same, which simplifies the translation scheme.

The translation of a case expression dispatching over a variant type to the underlying encoding transforms the overarching case into a chain of case-expressions. Finally, the variant declarations may be removed.

Examples: The simplest encoding corresponds to the type `Bool` of Boolean values:

$$\texttt{Bool} = \texttt{True} \mid \texttt{False} \Rightarrow 1 + 1$$

The `Maybe` datatype is encoded as:

$$\texttt{Maybe}\ \alpha = \texttt{Nothing} \mid \texttt{Just}\ \alpha \Rightarrow 1 + \alpha$$

While the encoding for generic lists exemplify most of the translation rules above simultaneously:

$$\texttt{List}\ \alpha = \texttt{Nil} \mid \texttt{Cons}\ \alpha\ (\texttt{List}\ \alpha) \Rightarrow \mu X.1 + A \times X$$

As an example of a translation of a case-expression using variants, we construct a variant for traffic lights where we leave irrelevant parts of the implementation undefined:

```
Lights = Red | Yellow | Green

case $e$ of
  Red ⇒ $c_1$
  Yellow ⇒ $c_2$
  Green ⇒ $c_3$
```

Is rewritten as:

```
case $e$ of
  inl(()) ⇒ $c_1$
  inr($e'$) ⇒ case $e'$ of
    inl(()) ⇒ $c_2$
    inr(()) ⇒ $c_3$
```

4.2 Type classes

Type classes in Haskell are aimed at solving overloading of operators by allowing types to implement or infer a class. A class is a collection of function names with accompanying type signatures, which are the functions to be inferred. We can use type classes to implement equality for example:

```
class Eq a where
  (==) ⇒ a → a → 1 ↔ Bool

instance Eq Nat where
  (==) $n_0$ $n_1$ () ⇒ eqInt $n_0$ $n_1$ ()

eqInt $n_0$: Nat $n_1$: Nat (): 1 = case unroll [Nat] $n_0$ of
  inl() = case unroll [Nat] $n_1$ of
    inl() ⇒ True
    inr($n_1'$) ⇒ False
  inr($n_0'$) = case unroll [Nat] $n_1$ of
    inl() ⇒ False
    inr($n_1'$) ⇒ eqInt $n_0'$ $n_1'$ ()
```

Where `Eq` is the type class name and `a` is a type variable which is substituted for a concrete type when the type class is instantiated. Notice the definition of the equality class member needs to keep both n_0 and n_1 as ancillae values as we cannot recover n_1 from the resulting boolean value alone.

The translation must include stripping away all definitions and instantiations of classes, and treat each class instantiations as top level function definitions. An obvious method would be to create unique functions for each instance that specialize for that instance type. Then any function f in which an overloaded function call takes place needs to be changed so that the function for a specific type is called instead. And since we need all functions to exist statically at run time, we need to generate a new definition of f for each type the class function is defined for in a cascading manner. Thus this translation is quite verbose, but it works as a proof of concept. We will present an implementation of the two presented ideas amongst others in future work.

5 Conclusion

Although CoreFun is a continuation of the work that was started with RFun, its abstract syntax and evaluation semantics are quite different and include more explicit primitive language constructs. However, we have also shown that CoreFun can be made lighter via syntactic sugar to mimic other functional languages.

We have presented a formal type system for CoreFun, including support for recursive types through a fix point operator and polymorphic types via parametric polymorphism. The type system is built on relevance typing, which is sufficient for reversibility if we accept that functions may be partial.

Evaluation has been presented through a big step semantics. Most evaluation rules were straight forward, but it was necessary to define a notion of leaves and a relation for "unification" as machinery to describe the side condition necessary to capture the first match policy.

An advantage offered by the type system is the ability to check the first match policy statically. A static guarantee that the first match policy should hold for a function will eliminate the run time overhead of case-expressions, often leading to more efficient evaluation. For simple types we can check for orthogonality of inputs and the possible values of leaf expressions. For recursive types, we need to apply an induction principle. However, it is difficult to detect exactly when this will yield a first match policy guarantee.

Finally, we have argued that it is possible to enhance the syntax of CoreFun with high level constructs, which in turn have simple translation schemes back to the core language. We have presented three examples, including variants and type classes, which, as an example, can be used to replace the duplication/equality operator in the original RFun language.

Future work will use CoreFun as the foundation for a modern-style reversible functional programming language. In contrast to many reversible programming the syntax of CoreFun does not have support for reverse application of functions. This is not problematic and the relational semantics does make it possible to inverse interpret a program. Thus the future language should also support this.

Acknowledgements This work was partly supported by the European COST Action IC 1405: Reversible Computation — Extending Horizons of Computing.

References

1. Anderson, A.R., Belnap, N.D.: Entailment: The logic of relevance and necessity, vol. 1. Princeton University Press (1975)
2. Axelsen, H.B., Glück, R.: On reversible turing machines and their function universality. Acta Informatica **53**(5), 509–543 (2016)

3. Dunn, J.M., Restall, G.: Relevance logic. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, vol. 6, pp. 1–192. Springer-Verlag, second edn. (2002)
4. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–101 (1987)
5. Glück, R., Kaarsgaard, R.: A categorical foundation for structured reversible flowchart languages. In: Silva, A. (ed.) Mathematical Foundations of Programming Semantics (MFPS XXXIII). Electronic Notes in Theoretical Computer Science, vol. 336, pp. 155–171. Elsevier (2018)
6. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: A scalable quantum programming language. In: Conference on Programming Language Design and Implementation, PLDI. pp. 333–342. PLDI '13, ACM (2013)
7. Huffman, D.A.: Canonical forms for information-lossless finite-state logical machines. IRE Transactions on Information Theory **5**(5), 41–59 (1959)
8. James, R.P., Sabry, A.: Theseus: A high level language for reversible computing (2014), work in progress paper at RC 2014. Available at www.cs.indiana.edu/~sabry/papers/theseus.pdf
9. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development **5**(3), 183–191 (1961)
10. Lecerf, Y.: Machines de Turing réversibles. Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences **257**, 2597–2600 (1963)
11. Lutz, C., Derby, H.: Janus: A time-reversible language. A letter to R. Landauer. `http://tetsuo.jp/ref/janus.pdf` (1986)
12. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
13. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
14. Polakow, J.: Ordered Linear Logic and Applications. Ph.D. thesis, Carnegie Mellon University (2001)
15. Sabry, A., Valiron, B., Vizzotto, J.K.: From symmetric pattern-matching to quantum control. In: Baier, C., Dal Lago, U. (eds.) Foundations of Software Science and Computation Structures. pp. 348–364. Springer International Publishing (2018)
16. Schordan, M., Jefferson, D., Barnes, P., Oppelstrup, T., Quinlan, D.: Reverse code generation for parallel discrete event simulation. In: Krivine, J., Stefani, J.B. (eds.) Reversible Computation, pp. 95–110. 9138, Springer (2015)
17. Schultz, U.P., Laursen, J.S., Ellekilde, L., Axelsen, H.B.: Towards a domain-specific language for reversible assembly sequences. In: Krivine, J., Stefani, J. (eds.) Reversible Computation, RC. vol. 9138, pp. 111–126 (2015)
18. Thomsen, M.K.: A functional language for describing reversible logic. In: Specification & Design Languages, FDL 2012. pp. 135–142. IEEE (2012)
19. Thomsen, M.K., Axelsen, H.B.: Interpretation and programming of the reversible functional language. In: Symposium on the Implementation and Application of Functional Programming Languages. pp. 8:1–8:13. IFL '15, ACM (2016)
20. Wadler, P.: Linear types can change the world! In: IFIP TC 2 Working Conference on Programming Concepts and Methods. pp. 347–359. North Holland (1990)
21. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) Reversible Computation, RC '11. LNCS, vol. 7165, pp. 14–29. Springer-Verlag (2012)
22. Yokoyama, T., Axelsen, H.B., Glück, R.: Fundamentals of reversible flowchart languages. Theoretical Computer Science (2015), dx.doi.org/10.1016/j.tcs.2015.07.046. Article in press
23. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Partial Evaluation and Program Manipulation. PEPM '07. pp. 144–153. ACM (2007)