Design of a Reversible Functional Language

And Its Type System

Petur Andrias Højgaard Jacobsen

Institute of Computer Science
University of Copenhagen

August 24, 2018

Table of Contents

Introduction

- Reversible computing is the study of computational models in which individual computation steps can be uniquely and unambiguously inverted.
- Originally studied for energy conservation, we wish to use it as an property of programming.
- Reversible languages include Janus, Pi, Theseus and RFun.
- We present a reversible functional programming language CoreFun, inspired by RFun.

Why is a Type System Useful?

- Well-typed programs cannot go wrong.
- Proper resource usage.
- Introduction of static information.
- Other?

CoreFun Introduction

CoreFun features:

- A relevant type system.
- Ancillae parameters.
- Rank-1 parametric polymorphism.
- Recursive types through iso-recursive treatment.

CoreFun Grammar

$$
\begin{aligned}
q &::= d^* \\
d &::= f\ \alpha^*\ v^+ = e \\
e &::= x \\
&\quad \mid () \\
&\quad \mid \mathbf{inl}(e) \\
&\quad \mid \mathbf{inr}(e) \\
&\quad \mid (e, e) \\
&\quad \mid \mathbf{let}\ l = e\ \mathbf{in}\ e \\
&\quad \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e, \mathbf{inr}(y) \Rightarrow e \\
&\quad \mid f\ \alpha^*\ e^+ \\
&\quad \mid \mathbf{roll}\ [\tau]\ e \\
&\quad \mid \mathbf{unroll}\ [\tau]\ e \\
l &::= x \\
&\quad \mid (x, x) \\
v &::= x : \tau_a
\end{aligned}
$$

Example Program: Map

Below is a rather standard `map` function. Note the explicit **roll** and **unroll** terms.

$$
\begin{aligned}
&\texttt{map}\ \alpha\ \beta\ (f : \alpha \leftrightarrow \beta)\ (xs : \mu X.1 + \alpha \times X) = \\
&\quad \texttt{case unroll}\ [\mu X.1 + \alpha \times X]\ \texttt{of} \\
&\qquad \texttt{inl}(()) \Rightarrow \texttt{roll}\ [\mu X.1 + \beta \times X]\ \texttt{inl}(()), \\
&\qquad \texttt{inr}(xs') \Rightarrow \texttt{let}\ (x, xs'') = xs' \\
&\qquad\qquad\qquad \texttt{in let}\ x' = f\ x \\
&\qquad\qquad\qquad \texttt{in let}\ xs''' = \texttt{map}\ \alpha\ \beta\ f\ xs'' \\
&\qquad\qquad\qquad \texttt{in roll}\ [\mu X.1 + \beta \times X]\ \texttt{inr}((x', x'''))
\end{aligned}
$$

CoreFun Typing Terms

Typing terms. τ_f are function types, τ are primitive types and τ_a are ancilla types.

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

Typing judgement

$$\Sigma; \Gamma \vdash e : \tau$$

Ensuring Relevance

Two rule for variables, depending on which context they appear in.

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

Unit requires an empty context. We can always perform Unit elimination.

$$\text{T-Unit:}\ \frac{}{\Sigma; \emptyset \vdash () : 1}$$

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

Ensuring Relevance

Introduce non-deterministic union of hypotheses when a rule has multiple premises.

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

Allows contraction and exchange. A concrete example:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \emptyset; x \mapsto \tau_x, y \mapsto \tau_y \vdash \ldots & \emptyset; y \mapsto \tau_y, z \mapsto \tau_z \vdash \ldots\end{array}}{\emptyset; x \mapsto \tau_x, y \mapsto \tau_y, z \mapsto \tau_z \vdash \ldots}$$

Ensuring Ancillae Are Static

We observe that hypotheses can be put into the static context when no dynamic information went into constructing them.

$$\text{T-Let1St:}\ \frac{\Sigma;\emptyset \vdash e_1 : \tau' \qquad \Sigma, x : \tau';\Gamma,\vdash e_2 : \tau}{\Sigma;\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

Alternatively, We could have gone with restricted weakening.

$$\text{T-Weakening:}\ \frac{\Sigma, x \mapsto \tau;\Gamma \vdash e : \tau'}{\Sigma, x \mapsto \tau;\Gamma, x \mapsto \tau \vdash e : \tau'}$$

Evaluation

Big step call-by-value semantic with substitution to map values to variables. Evaluation judgement

$$p \vdash e \downarrow c$$

The difficult part of ensuring reversibility is ensuring determinism of branching.

We employ a *First Match Policy*. It can be seen in the E-CaseR rule.

Now, we define *leaf expressions*, *possible leaf values* and *leaves*.

Leaf Expressions, Possible Leaf Values and Leaves

Leaf expressions are a subset of expression which may be regarded as the final expression.

We collect the set of leaves of an expression:

$$\mathsf{leaves}(\textbf{case } z \textbf{ of } \textbf{inl}(x) \Rightarrow x, \textbf{inr}(y) \Rightarrow y) = \{x, y\}$$

A unification of two expressions states if the expressions relate to one another. An example:

$$() \rhd ()$$

The set of possible leaf values is then defined as:

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

Backwards Determinism

Theorem (Contextual Backwards 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 $1 \leq i \leq n$.

Normally expressed directly by the reduction relation, as in RFun. Too restrictive in CoreFun ($e \downarrow c$ and $e' \downarrow c \Leftrightarrow e = e'$ is only true in trivial languages).

First Match Policy Cons

Unfortunately, reversibility cannot be fully determined by the type system.

The First Match Policy is potentially inefficient! See the following plus function:

$$
\begin{aligned}
&\texttt{plus}\ (n_1 : \mu X.1 + X)\ (n_2 : \mu X.1 + X) = \\
&\quad \texttt{case unroll}\ [\mu X.1 + X]\ n_1\ \texttt{of} \\
&\qquad \texttt{inl}(()) \Rightarrow n_2, \\
&\qquad \texttt{inr}(n_1') \Rightarrow \texttt{let}\ n_2' = \texttt{plus}\ n_1'\ n_2 \\
&\qquad\qquad\qquad \texttt{in roll}\ [\mu X.1 + X]\ \texttt{inr}(n_2')
\end{aligned}
$$

Alternative Measures: Exit Assertion

We will add two things: Exit assertions and static guarantees.

An exit assertion is appended to a case-expression and must evaluate to a Boolean value.

- Janus if-statement:

$$\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi } e_a$$

- CoreFun case-expression with exit assertion:

$$\textbf{case } e \textbf{ of } \textbf{inl}(x) \Rightarrow e_2, \textbf{inr}(y) \Rightarrow e_3 \textbf{ safe } e_a$$

Alternative Measures: Static Guarantee

Two methods to identify a static guarantee:

- "Traditional" analysis looks at the open form of the branch arms to observe syntactic difference

$$\{(l_2, l_3) \mid l_2 \in \mathsf{leaves}(e_2), l_3 \in \mathsf{leaves}(e_3), l_2 \rhd l_3\} = \emptyset$$

- Exhaustive method looks at the domain of a function to see if any leaves sets have overlap.

$$\mathsf{leaves}_l = \bigcup_{\substack{(c_1,\ldots,c_n) \\ \in \mathsf{dom}(f)}} \mathsf{leaves}(e_2')$$

$$\mathsf{leaves}_r = \bigcup_{\substack{(c_1,\ldots,c_n) \\ \in \mathsf{dom}(f)}} \mathsf{leaves}(e_3')$$

$$\{(l_2, l_3) \mid l_2 \in \mathsf{leaves}_l, l_3 \in \mathsf{leaves}_r, l_2 \rhd l_3\} = \emptyset$$

Running Backwards

The inverse of any function can be determined by trying all possible outputs, known as the Generate-And-Test method.

We want it to be efficient though: Meaning, as efficient as forward interpretation.

Program Inversion

A program inverter literally writes the inverse program from the forward program. This program can then be evaluated as normal.

$$\mathcal{I}_e[\![e]\!] = e'$$

It is presented as a set of local inverters, one for each syntactic domain. For CoreFun we could define $\mathcal{I}_e, \mathcal{I}_f, \mathcal{I}_p$.

This is easy for flow-chart languages like Janus, as constructs are symmetrical and have clear inverses.

Not so easy for reversible functional languages.

Inverse Semantics

We will instead use inverse interpretation.

Relationship between a function f and its inverse:

$$f\ a_1 \ldots a_n\ x = y \iff f^{-1}\ a_1 \ldots a_n\ y = x$$

Cannot describe mapping of variables to values with substitution. We instead "predict" the value of each variable in the forward direction. Inverse semantics judgement

$$p; \sigma \vdash^{-1} e, c \rightsquigarrow \sigma'$$

Inverse Semantics Correctness

The derivation of the forward semantics and the backwards semantics are captured by the following lemma:

Lemma

If $p \vdash e[c/x] \downarrow c'$ *(by* $\mathcal{E}$*) where* x *is the only free variable in* e*, then* $p; \emptyset \vdash^{-1} e, c' \rightsquigarrow \{x \mapsto c\}$ *(by* $\mathcal{I}$*).*

We use it directly to argue the correctness of the inverse semantics:

Theorem (Correctness of Inverse Semantics)

If $p \vdash f\ c_1 \ldots c_n \downarrow c$ *with* $p(f) = \alpha_1 \ldots \alpha_m\ x_1 \ldots x_n = e$*, then* $\sigma(x_n) = c_n$ *where* $p; \emptyset \vdash^{-1} e[c_1/x_1, \ldots, c_{n-1}/x_{n-1}], c \rightsquigarrow \sigma$.

Inverse Semantics Expressiveness

A program inverter is less expressive than the presented inverse semantics. Consider inverting:

$$\mathcal{I}_f[\![f\ (x:\tau) = (x,x)]\!] \stackrel{\text{def}}{=} f^{-1}\ (x':\tau\times\tau) = \mathcal{I}_e[\![x']\!]$$

$$\mathcal{I}_e[\![x']\!] \stackrel{\text{def}}{=} \textbf{let}\ (x,y) = x'\ \textbf{in}\ x$$

We cannot implicitly assume that $x = y$, so program is not well-typed. But we can run backwards by the inverse semantics.

Transformations Background

CoreFun is as simple as possible. Our next motivation is to lift it into a higher-level language to make programming easier.

Transformations should have clearly defined translations of expressions from a high-level syntax to the core syntax.

$$e \overset{\text{def}}{=} e'$$

Where e is an expression in the light language and e' is an expression in the core language.

Variants

Allows us to define new algebraic data types.

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

The translation strips all data type definitions and substitutes directly into each expression.

Translations of case-expressions over variants unroll the variant.

Variant Translation Example

Light Program containing a variant:

$$\texttt{Choice} = \texttt{Rock} \mid \texttt{Paper} \mid \texttt{Scissors}$$

$$\begin{aligned}
&\texttt{rpsAI}\ (c{:}\texttt{Choice}) = \\
&\quad \texttt{case}\ c\ \texttt{of} \\
&\qquad \texttt{Rock} \Rightarrow \texttt{Paper} \\
&\qquad \texttt{Paper} \Rightarrow \texttt{Scissors} \\
&\qquad \texttt{Scissors} \Rightarrow \texttt{Rock}
\end{aligned}$$

Variant Translation Example

Translated core program:

```
rpsAI (c:1 + (1 + 1)) =
  case c of
    inl(()) ⇒ inr(inl(()))
    inr(w) ⇒ case w of
      inl(()) ⇒ inr(inr(()))
      inr(()) ⇒ inl(())
```

Type Classes

Type classes are famous from Haskell. They solve overloading of operations. A type class definition:

$$\textbf{class } \kappa\ \alpha \textbf{ where } [f \Rightarrow \tau_f]^+$$

Functions which use some type class operator have their signature restricted:

$$f\ \kappa\ \alpha \Rightarrow \alpha\ .\ (x_1 : \tau_1) \ldots (x_n : \tau_n) = e$$

The translation erases all type class definitions and generates new top-level functions.

And the correct top-level functions is applied instead of the generic function.

Type Classes

Can be used to define a duplication/equality operator (with variants):

$$\texttt{Eq}\ \alpha = \texttt{Eq} \mid \texttt{Neq}\ \alpha$$

Type Class Translation Example

Light Program containing a type class:

```
class Default a where
  def ⇒ 1 ↔ a

instance Default (μX.1 + X) where
  def u ⇒ inl(())

defPair : Default α, Default β ⇒ α β . 1
defPair u = (def α u, def β u)
```

Type Class Translation Example

Translated core program:

$$\texttt{defNat} : 1$$
$$\texttt{defNat}\ u = \texttt{inl}(())$$

$$\texttt{defPairNatNat} : 1$$
$$\texttt{defPairNatNat}\ u = (\texttt{defNat}\ u, \texttt{defNat}\ u)$$

Top-level Cases

Functions oftentimes start with looking at the form of the input. We introduce a shorthand for this phenomenon.

$$\texttt{map}\ \ f\ \ \texttt{[]} = \texttt{[]}$$
$$\texttt{map}\ \ f\ \ (x : xs) = f\ \ x : \texttt{map}\ \ f\ \ xs$$

Only allow top-level sum types and variant types. The coverage must be *total*.

Requires constructing a tree, as we allow it for multiple parameters at the time.

Top-level Case Translation Example

Light Program containing a top level clause and a list variant:

$$
\begin{aligned}
&\texttt{map} :: \ \alpha\ \beta\ .\ (\alpha \leftrightarrow \beta) \rightarrow \texttt{List}\ \ \alpha \\
&\texttt{map}\ f\ \texttt{inl}(()) = \texttt{roll}\ [\texttt{List}\ \beta]\ \texttt{Nil} \\
&\texttt{map}\ f\ \texttt{inr}((x, xs')) = \texttt{let}\ x' = f\ \alpha\ \beta\ x \\
&\qquad\qquad\qquad\qquad\qquad\quad \texttt{in let}\ xs'' = \texttt{map}\ \alpha\ \beta\ f\ xs' \\
&\qquad\qquad\qquad\qquad\qquad\quad \texttt{in roll}\ [\texttt{List}\ \beta]\ (\texttt{Cons}\ x'\ xs'')
\end{aligned}
$$

Guards

Guards are built on top of top-level clauses and require a predicate to hold on top of the form of the pattern expressions.

We require guards to be *total*. One clause should be guarded by an **otherwise**.

They are easily translated into a chain of boolean checks, one for each guard, in order of definition.

Guard Translation Example

Light Program containing guards.

$$\begin{aligned}
&\texttt{tryPred} :: \mu X.1 + X \\
&\texttt{tryPred x} \mid \texttt{case unroll } [\mu X.1 + X]\ x \texttt{ of} \\
&\qquad\qquad \texttt{inl}(()) \Rightarrow \texttt{inl}(()), \\
&\qquad\qquad \texttt{inr}(x') \Rightarrow \texttt{inr}(()) = \texttt{inl}(()) \\
&\texttt{tryPred x} \mid \texttt{otherwise} = \texttt{let } x' = \texttt{unroll } [\mu X.1 + X]\ x \\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{in inr}(x')
\end{aligned}$$

Guard Translation Example

Translated core program.

$$\texttt{tryPred} :: \mu X.1 + X$$

$$\begin{aligned}\texttt{tryPred x} = \texttt{case unroll } [\mu X.1 + X]\ x \texttt{ of}\\ \texttt{inl}(()) &\Rightarrow \texttt{inl}(())\\ \texttt{inr}(()) &\Rightarrow \texttt{let } x' = \texttt{unroll } [\mu X.1 + X]\ x\\ &\quad\ \texttt{in inr}(x')\end{aligned}$$

Records

Records are labeled products (and dual to variants).

$$\gamma = \{l_1 :: \tau_1, \ldots, l_n :: \tau_n\}$$

We require them to be total and we disallow projections. Instead, we introduce a special projection scope:

$$\textbf{within } \gamma : e^{\rho} \textbf{ end}$$

Translations of introductions of records construct an n-ary product directly.

A record scope is a chain of let-expressions record and introduce

Record Translation Example

Light Program containing records.

$$\texttt{Vector} = \{\ \texttt{x}\colon \mu X.1 + X,\ \texttt{y}\colon \mu X.1 + X,\ \texttt{z}\colon \mu X.1 + X\ \}$$

$$\begin{aligned}
&f\ (x : \mu X.1 + X)\ (y : \mu X.1 + X)\ (z : \mu X.1 + X) = \\
&\quad \texttt{let}\ v = \texttt{Vector}\ \{\ \texttt{x} = x,\ \texttt{y} = y,\ \texttt{z} = z\ \} \\
&\quad \texttt{in within}\ v\colon \\
&\qquad\quad v.x = \texttt{roll}\ [\mu X.1 + X]\ \texttt{inr}(v.x) \\
&\qquad \texttt{end}
\end{aligned}$$

Record Translation Example

Translated core program.

$$
\begin{aligned}
&f\ (x : \mu X.1 + X)\ (y : \mu X.1 + X)\ (z : \mu X.1 + X) = \\
&\quad \texttt{let}\ v = (x, (y, z)) \\
&\quad \texttt{in let}\ x' = \texttt{roll}\ [\mu X.1 + X]\ \texttt{inr}(x) \\
&\qquad \texttt{in}\ (x', (y, z))
\end{aligned}
$$

Misc

In the thesis, we presented two more transformations, not worthy of a lengthy discussion:

- Arbitrarily sized products.
- Multiple let-expressions in a row.

Both of these have straightforward translations by unfolding.

Anonymous Functions and Binary Operators

Let us consider two translations which are not very amenable: anonymous functions and binary operators.

- Anonymous functions are tricky as we need to take them out of their lexical scope. What information do they use?
- Binary operators are tricky as we transform the right operand, but this does not fit well with what is expected:

$$a \odot b = c \qquad 2 + 3 = 5$$
$$a \odot^{-1} c = b \qquad 2 - 5 = 3$$

Higher Order Programming

Remember that we want to design a garbage-free language.

Supporting general higher-order functions means variables can take on arbitrary functions as values.

Consider the following function:

$$g\ (x : \tau) = \textbf{let}\ f' = \texttt{rtwice}\ \ f\ \textbf{in}\ f'\ x$$

How is $f'\ x$ computed? The function f' is unknown during inverse evaluation. Therefore this generates a *closure*.

Introduction†

We set out to add a type system to RFun, but ended up designing a new language with a relevant type system.

We showed evaluation rules and backwards determinism.

We showed that it is possible to inverse interpret CoreFun programs.

We showed syntactic sugar over the core language to make CoreFun more modern.

The work in this thesis has also been made into an article.