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.

```
data Nat = Z | S Nat

plus :: Nat → Nat ↔ Nat
plus Z     x = x
plus (S y) x =
  let x' = plus y x
   in (S x')
```

```
fib :: Nat ↔ (Nat, Nat)
fib Z     = ((S Z), Z)
fib (S m) =
  let (x, y) = fib m
       y'    = plus x y
  in  (y', x)
```

Figure 1: RFun program computing Fibonacci pairs.

```
map :: (a ↔ b) → [a] ↔ [b]
map fun [ ]  = [ ]
map fun (l:ls) =
  let l'  = fun l
      ls' = map fun ls
  in  (l': ls')
```

Figure 2: Map function in RFun.

1.1 Type system

With Milner's motto that "well-typed programs cannot go wrong," type systems have proven immensely successful in guaranteeing fundamental well-behavedness properties of programs. In reversible functional programming, linear type systems (see, *e.g.*, [2]) have played an important role in ensuring reversibility.

Fundamentally, a reversible computation can be considered as an injective transformation of a state into an updated state. In this view, it seems obvious to let the type system guarantee linearity, *i.e.*, that each available resource (in this case, variable) is used exactly once. Though linearity is not enough to guarantee reversibility, it enables the type system to statically reject certain irreversible operations (*e.g.*, projections). However, linearity is also more restrictive than needed: if we accept that functions may be partial (a necessity for r-Turing completeness), first-order data *can* be duplicated reversibly. For this reason, we may relax the linearity constraint to *relevance*, *i.e.*, that all available variables must be used *at least* once. This guarantees that values are never lost, while also enabling implicit duplication of values.

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. For example, in Figure 1, the first input variable of the `plus` function is ancillary; it's value is tacitly returned automatically as part of the output.

To support such ancillary variables at the type level, a type system inspired by Polakow's combined reasoning system of ordered, linear, and unrestricted intuitionistic logic [4] 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 the dynamic one.

Whereas we must ensure that variables in the dynamic context Γ are used in a relevant manner to guarantee reversibility, there are no restrictions on the use of variables in the static context – these can used as many or as few times (including not at all) as desired. To distinguish between ancillary and dynamic variables at the type level, two different arrow types are used: $t_1 \rightarrow t_2$ denotes that the input variable is ancillary, whereas $t_1 \leftrightarrow t_2$ denotes that it is dynamic. As such, the type of `plus` in Figure 1 signifies that the first input variable is ancillary, and the second is dynamic.

A neat use of ancillae is to provide limited support for behaviour similar to higher-order functions. For example, the usual `map` function is not reversible, as not every function of type `[a]` $\leftrightarrow$ `[b]` arises as a functorial application of some function of type `a` $\leftrightarrow$ `b`. However, if we consider the input function `f` :: `a` $\leftrightarrow$ `b` to be ancillary, one can straightforwardly define a reversible `map` function (see Figure 2) as one of type (`a` $\leftrightarrow$ `b`) $\rightarrow$ (`[a]` $\leftrightarrow$ `[b]`). In this way, ancillae can be considered as a slight generalization of the *parametrized maps* found in Theseus [2].

1.2 Duplication/Equality

In the first version of RFun, duplication and equality was included as a special operator, which could perform deep copying or uncopying reversibly, depending on the usage. However, we have found that understanding the semantics this operator often poses a problem for programmers.

To remedy this, we propose to use type classes, and implement equality instead using a type class similar to the `EQ` type class found in Haskell. As in Haskell, the functions needed to be member of this class can often be automatically derived.

1.3 First-match policy

The first-match policy (FMP) is essential to ensuring injectivity of individual functions. It states that a returned value of a function *must not* match any previous leaf of the function, and can be compared to checking the validity of an assertion on exit.

I the first version of RFun, the check to ensure that the first-match policy was upheld was always performed at run-time, and, thus, posed a limitation to the performance. However, with the type system, it will now often be possible to perform this check statically, as the types of the leaves or even the ancillae inputs can be orthogonal. *E.g.* in the `plus` function (in Figure 1), the use of ancillae input ensures that the FMP is always upheld, while `map` (in Figure 2) is reversible by orthogonality of the leaves. Unfortunately, this cannot always guaranteed statically, and the `fib` function (in Figure 1) is an example where a runtime check is still required.

1.4 Conclusion

In this paper we have outlined the future development of the reversible function language RFun. A central element of this is the development of the type system. The work shows both an interesting new application of relevant type systems, and gives RFun a more modern design that will make it easier for programmers to understand.

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

References

[1] D. A. Huffman. Canonical forms for information-lossless finite-state logical machines. *IRE Transactions on Information Theory*, 5(5):41–59, 1959.

[2] R. P. James and A. Sabry. Theseus: A high level language for reversible computing. Work in progress paper at RC 2014. Available at `www.cs.indiana.edu/~sabry/papers/theseus.pdf`, 2014.

[3] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[4] J. Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001.

[5] M. K. Thomsen and H. B. Axelsen. Interpretation and programming of the reversible functional language. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, pages 8:1–8:13. ACM, 2016.

[6] T. Yokoyama, H. B. Axelsen, and R. Glück. Towards a reversible functional language. In A. De Vos and R. Wille, editors, *Reversible Computation, RC '11*, volume 7165 of *LNCS*, pages 14–29. Springer-Verlag, 2012.