Project Proposal

Concepts of Program Design

Joris ten Tusscher, Joris Burgers, Ivo-Gabe de Wolff, Cas van der Rest, Orestis Melkonian

1 Background

1.1 Logical and Physical Reversibility

The notion of reversible computations can generally be broken down in two main categories: *physical reversibility* and *logical reversibility.* The former concerns the usage of physically reversible processes to carry out computations and has theoretical applications in the creation of more cost-efficient hardware[1]. Logical reversibility is the study of computations that can be *undone*, i.e. we can reconstruct the initial state of a computation solely from it's result and the computation itself. Reversible programming languages are part of the domain of logical reversibility.

1.2 Computational Expressiveness

Reversible Turing machines can be used to define the computational expressiveness of programs written in reversible languages. They differ from regular deterministic Turing machines in that they are not only forward deterministic, but also backward deterministic. See [2] for the exact definition of reversible Turing machines. (Reversible) programming languages that are able to simulate any reversible Turing machine are said to be r-Turing complete.

This means that, when we think about reversible programming languages on a more abstract level, any function describing an atomic transition within the program's state space has to be injective. This imposes some practical limitations on the design of reversible programming languages. Multiplication and assignment are examples that are found in most programming languages and are non-injective, meaning that they cannot be incorporated into reversible programming languages in the same way as in regular programming languages.

1.3 Reversible Programming Languages

Notable attempts at designing reversible programming languages include the language *Janus*[3]. An example of how Janus deals with the previously described limitations is that it only allows combined assignment and modification of variables with the +=, -= and ^= operators. The result is a language that quite nicely demonstrates the concept of reversible programming languages, but is simultaneously fairly unpractical due to it's reversible nature. The semantics of Janus were later formalized by Yokoyama[9]

Other examples of proposed reversible programming languages include the functional language RFUN [4][5], Inv[6], Arrow[7] and the object oriented language Joule[8].

2 Problem

2.1 Experiment

2.2 Janus

We want to implement the reversible programming language Janus as an DSL in Haskell. The syntax and semantics given by Lutz en Derby [**?**] will be used. There will however be a few changes. There will be no input and output in the DSL. Therefore, the READ and WRITE statements will not be valid statements. Besides this limitation, the DSL will allow arbitrary Haskell expressions to the right-hand side of Janus statements, enabling programmers to reuse Haskell code. This freedom of accepting Haskell code will not limit the reversibility of the DSL, since the semantics for Janus require statements to be reversible, but not right-hand expressions. For instance, the statement $x+ = f$ where f is any valid Haskell expression. As long as f does not depend on x, the statement can be inverted by the statement $x- = f$.

2.3 Extensions

- Data structures (trees/sets/maps/ADTs)
- Prelude
- Syntactic sugar

2.4 Applications

- Path finding algorithms
- Encoding-Decoding
- Low-level bit manipulation (hamming error correction)
- [Optional] Reversible Debugger

2.5 Comparison with Existing Implementations

2.5.1 Benchmarking

We intend to compare the performance of our embedded DSL to existing implementations of the Janus language. We will do so by implementing the same algorithms in both our DSL and standard Janus, and comparing the time it takes the different implementations to compile and run the programs, as well as memory usage.

2.5.2 Other Metrics

Since our main goal is to improve the usability of Janus, it is hard to come up with metrics that provide a useful insight in the quality of our result. Certainly, performance alone will not be enough to capture the differences between the embedding and existing implementations. In order to attempt to measure the quality of our result beyond raw performance, we will compare our results with existing implementations with respect to the following aspects as well.

- LOC when implementing a certain algorithm
- The amount of language constructs and abstractions at a programmer's disposal when solving problems
- The ability to use patterns/constructs that are generally considered as idiomatic or elegant within the programming community

Admittedly, most of the above comes down to opinion. We think that it is useful to think about these aspects when considering our results nonetheless.

2.6 Formal verification

To our knowledge, there is no formally verified interpreter for Janus. The most interesting propositions to prove about an implementation of Janus are the following:

- r-Turing completeness
- Automated verification of conditional and loop assertions
- Reversibility of Janus programs

We aim on formally verifying the reversibility of Janus programs, because it appears to be the most feasible option, while being as meaningful as the rest.

3 Methodology

3.1 Template Haskell

In order to embed Janus as a DSL in Haskell, we are going to use the *TemplateHaskell*(TH) and *QuasiQuotation* GHC extensions, which extend Haskell with compile-time meta-programming capabilities. This will allow us to perform static checking (i.e. syntactic and semantic checking), code generation and embedding Haskell expressions in Janus at compile-time.

Our main focus will be on static checking, as we are not planning to generate low-level efficient code for running our Janus programs. Nonetheless, we plan on using compile-time code generation to automatically derive the inverse of each declared Janus function and we expect this to give significant performance gains, since at the evaluation of an **uncall** statement, we will bypass the elaborate procedure of reversing the given function, whose computation time depends on its size.

On the static checking side of things, TH will allow us to perform the following at compile-time:

- Parsing
- Checking programs are well-typed
- Proper variable usage
- Embedding of Haskell expressions

Specifically, Janus programs will be written as *quasi-quote* strings inside a Haskell source file, which will eventually transfer control to a *quasi quoter*, that will parse and statically check desired properties of the given program, as well as compile the domain-specific syntax to Haskell code.

For instance, Janus statements of the form **id** ⟨REVERSIBLEOP⟩ **expr** require that the identifier on the left-hand side does not occur in the expression on the right-hand side. This is easily achieved in TH via the process of *reification*, which allows us to query compile-time information (e.g. a variable's type and identifier) while running our meta-programs.

In addition to compile-time guarantees, the embedded Janus language will also inherit powerful features, already existent in Haskell (e.g. modularity via Haskell's modules).

3.2 Benchmarking

- GHC Profiling
- Criterion package[1]

3.3 Formal Verification

In order to verify the reversibility of Janus programs, we will use *LiquidHaskell*, which extends Haskell with refinement types (i.e. types accompanied by logical predicates that enforce certain properties). *LiquidHaskell* will enable us to write Haskell functions, which will act as equation proofs of the reversibility of all possible Janus programs.

Mechanically checking the above statement can be achieved by performing structural induction on the syntax of our DSL. Technically, this proof will correspond to a Haskell function, which will perform recursion on the data types of our AST, via the process of *refinement reflection* (i.e. reflecting the code of a function in its output type).

3.4 Risk and contingency plans

There are a number of risks to consider when creating a DSL, which are specified below with their respective contingency plans.

- **Arbitrary types** The plan is to implement arbitrary types in the DSL. It is expected that implementing arbitrary types will take a long time and therefore, the risk exists that the implementation can not

[1]http://hackage.haskell.org/package/criterion

be completed in the given timespan. The contingency plan, if this goal takes too much time, is to reduce arbitrary types to some special types.

- **Full verification** The goal is to completely verify the implementation of the DSL. To our knowledge, there is, to our knowledge, no verified implementation of a reversible language. Therefore, we are aiming at providing a fully verified implementation. If this full verification is not possible, the contingency plan is to at least verify the basic, non-extended version of Janus.

- **Impossible applications** It is a possibility that the applications that will be implemented in the DSL are impossible to implement in a reversible language. If there is an application where there is no suitable variant found that can be implemented in a reversible language, that specific application will be ignored or a suitable replacement will be looked for.

- **Impossible use of libraries** There is the risk that one of the proposed libraries we plan to use to implement the DSL turns out not to be suitable. This may be the case because the library lacks some features that are necessary for the implementation of the DSL. If this is the case, a replacement will be looked for. If no suitable replacement can be found, the functionality will either be implemented in another way without the use of a library or will be removed from the specification.

3.5 Goal

4 Planning

- Milestones

- Division of labour

References

[1] Frank, M. P. (2005, May). Introduction to reversible computing: motivation, progress, and challenges. In Proceedings of the 2nd Conference on Computing Frontiers (pp. 385-390). ACM.

[2] Axelsen, H. B., & Glück, R. (2011, March). What do reversible programs compute? In FoSSaCS (pp. 42-56).

[3] Lutz, C., & Derby, H. (1982). Janus: a time-reversible language. Caltech class project.

[4] Yokoyama, T., Axelsen, H. B., & Glck, R. (2011). Towards a Reversible Functional Language. RC, 7165, 14-29.

[5] Thomsen, M. K., & Axelsen, H. B. (2015, September). Interpretation and programming of the reversible functional language RFUN. In Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages (p. 8). ACM.

[6] Mu, S. C., Hu, Z., & Takeichi, M. (2004, July). An injective language for reversible computation. In MPC (pp. 289-313).

[7] Rose, E. (2015). Arrow: A Modern Reversible Programming Language (Doctoral dissertation, Oberlin College).

[8] Schultz, U. P., & Axelsen, H. B. (2016, July). Elements of a Reversible Object-Oriented Language. In International Conference on Reversible Computation (pp. 153-159). Springer International Publishing.

[9] Yokoyama, T. (2010). Reversible computation and reversible programming languages. Electronic Notes in Theoretical Computer Science, 253(6), 71-81.

WEEK 1	**Task 1:** Orestis,... **Task 2:** Cas,...
WEEK 2	**Task 3:** Joris1, Joris2,... **Task 4:** Ivo,...
WEEK 3	**Task 1:** Orestis,... **Task 2:** Cas,...
PROGRESS REPORT	
WEEK 4	**Task 3:** Joris1, Joris2,... **Task 4:** Ivo,...
WEEK 5	**Task 1:** Orestis,... **Task 2:** Cas,...
WEEK 6	**Task 3:** Joris1, Joris2,... **Task 4:** Ivo,...
PROJECT SUBMISSION	

References

[1] Frank, M. P. (2005, May). Introduction to reversible computing: motivation, progress, and challenges. In Proceedings of the 2nd Conference on Computing Frontiers (pp. 385-390). ACM.

[2] Axelsen, H. B., & Glück, R. (2011, March). What do reversible programs compute? In FoSSaCS (pp. 42-56).

[3] Lutz, C., & Derby, H. (1982). Janus: a time-reversible language. Caltech class project.

[4] Yokoyama, T., Axelsen, H. B., & Glck, R. (2011). Towards a Reversible Functional Language. RC, 7165, 14-29.

[5] Thomsen, M. K., & Axelsen, H. B. (2015, September). Interpretation and programming of the reversible functional language RFUN. In Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages (p. 8). ACM.

[6] Mu, S. C., Hu, Z., & Takeichi, M. (2004, July). An injective language for reversible computation. In MPC (pp. 289-313).

[7] Rose, E. (2015). Arrow: A Modern Reversible Programming Language (Doctoral dissertation, Oberlin College).

[8] Schultz, U. P., & Axelsen, H. B. (2016, July). Elements of a Reversible Object-Oriented Language. In International Conference on Reversible Computation (pp. 153-159). Springer International Publishing.

[9] Yokoyama, T. (2010). Reversible computation and reversible programming languages. Electronic Notes in Theoretical Computer Science, 253(6), 71-81.