Hanus: The embedded reversible language in Haskell

Report - Concepts of Program Design

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

1 Problem

1.1 Experiment

In Janus, a time-reversible programming languages, all procedures can be executed forward and backward. This gives a limitation on the programs that a programmer can write and makes it thus hard to write certain applications. That gave us the following research question for our experiment:

> *Can we improve the usability of Janus by implementing it as a DSL embedded in Haskell?*

In order to be able to answer this question, we attempted to improve Janus in several ways, mainly by incorporating various elements of the host language (Haskell) in the DSL, as well as the addition of certain language constructs. We implemented certain algorithms in both our DSL and standard Janus and compared the implementations with respect to various metrics, such as performance and readability.

In this report, we will describe our implementation of the Janus as a DSL in Haskell, which we called *Hanus*. We will start by giving some context on Janus and reversible languages in general in section 1. We will describe the extensions and expected outcome in section 2. We describe on the results of our DSL in section 3 and reflect on the process in section 4.

1.2 Janus

We implemented the reversible programming language Janus as a DSL in Haskell. The syntax and semantics given by Lutz en Derby [1] were 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

1.3 Extensions

We added various extensions, mainly focused on improving the usability of our DSL:

- Local variable blocks
- Function arguments

- Arbitrary types

Some Janus features, such as the stack and arrays will not be implemented directly into the DSL, but are available through predefined functions in the host language.

1.4 r-Turing completeness

Given that only reversible programs can be written in a reversible language, it can only compute injective functions. This means that it cannot simulate any Turing machine, as Turing machines can compute non-injective functions. Thus, reversible languages are not Turing complete [2].

Axelsen e.a. [2] proposed the notion of reversible Turing completeness (r-Turing completeness), meaning that the language can simulate any reversible Turing machine (RTM). To define this, we must first know the concepts of forward and backward determinism.

- A Turing machine is *forward deterministic* if from any state and tape, only one transition to a new state and tape exists.
- Similarly, a Turing machine is *backward deterministic* if given any state and tape, only one transition that leads to this state exists.

A reversible Turing machine is defined as a forward and backward deterministic Turing machine.

1.4.1 Convert Turing machines

If we want to compute some non-injective function $f(x)$, we can create an injective function $x \to (f(x), x)$. As shown by [2], if $f(x)$ is computable by some Turing machine, then there exists a reversible Turing machine which computes $x \to (f(x), x)$ which uses three memory tapes instead of one. This three-tape reversible Turing machine can be converted to a normal single tape reversible Turing machine.

Given a computable function f, we can thus write a program in a reversible language that computes $x \to (f(x), x)$. However, this might use more memory and time than in a non-reversible language.

In Hanus, this trick can be used with only a minimal amount of additional memory and time. For simplicity, assume that f is a function from integers to integers. When function f is implemented in Haskell, a Hanus procedure that computes $(0, x) \to (f(x), x)$ can be created as follows.

```
procedure g(y :: Int, x :: Int) {
  y += f x;
}
```

2 Methodology

2.1 DSL Embedding

In order to embed Janus as a DSL in Haskell, we used the *TemplateHaskell*(TH)[3] and *QuasiQuotation*[4] GHC extensions, which extend Haskell with compile-time meta-programming capabilities. This allows us to perform static checking (i.e. syntactic and semantic checking), code generation and embedding Haskell expressions in Janus at compile-time.

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

- Parsing
- Checking that programs are well-typed
- Verifying proper variable usage
- Embedding of Haskell expressions

Specifically, Hanus programs can 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, Hanus 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 Hanus language will also inherit powerful features, already existent in Haskell (e.g. modularity via Haskell's modules).

2.2 Parser

We implemented the parser for Hanus using uu-parsinglib[1]. This allowed us to write a parser using parser combinators and get error correction. However, the error correction was not very reliable and can cause that the program does not terminate if there is a syntactic error. We expect that this is caused by the way that we parse Haskell expressions.

2.2.1 Parsing Haskell expressions and types

The syntax for our DSL supports Haskell expressions and types. We use the haskell-src-meta package[2] to parse those. However, to use this package, we must know where a Haskell expression or type ends. Consider an assignment of the form x += *expr;*, where `expr` is some expression. We will find the first occurrence of a semicolon in the source

[1] https://hackage.haskell.org/package/uu-parsinglib
[2] https://hackage.haskell.org/package/haskell-src-meta

string and try to parse this section of the source. If the source code is `x += length "a;b";`, it will try to parse `length "a` as a Haskell expression. This will fail, and the parser will try to find the next occurrence of a semicolon. Thus, it will try to parse `length "a;b"`, which does parse.

We expect that this does not play well with the error correction mechanisms of uu-parsinglib and cause that the parser does not terminate on invalid inputs. However, on some inputs the error correction does correct the input.

2.3 Expected outcome

2.4 Planning

3 Results

3.1 Achievements

3.2 Goal/planning adjustments

4 Reflection

4.1 Good/bad surprises

4.2 Problems along the way

5 Appendix

5.1 Repo link

5.2 Code navigation

References

[1] C. Lutz and H. Derby, "Janus: a time-reversible language," *Caltech class project*, 1982.

[2] H. B. Axelsen and R. Glück, "What do reversible programs compute?," in *FoSSaCS*, pp. 42–56, Springer, 2011.

[3] T. Sheard and S. P. Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, (New York, NY, USA), pp. 1–16, ACM, 2002.

[4] G. Mainland, "Why it's nice to be quoted: quasiquoting for haskell," in *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pp. 73–82, ACM, 2007.