Project 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 [?] 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 [?].

Axelsen e.a. [?] 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 [?], 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)[?] and *QuasiQuotation*[?] 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 Expected outcome

2.3 Planning

3 Results

3.1 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.

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

3.1.1 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 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.

3.1.2 Semantics of operators

In Hanus, a user is able to define their own types and operators. These operators can be defined for their own types or for types that are already defined in Haskell. In Hanus, there are already a number of operators and types defined that users can use.

3.1.2.1 Definition of arbitrary types

In a reversible language, any variable should have a default value with which a variable is initialised before the *main* procedure is run. The original *Janus* specification only allowed for integers. For integers the default value of 0 is chosen. In Hanus, for any number of the *Haskell Num* class, the default is also 0. There are also a number of other types predefined in Hanus. Their defaults are implemented as follows:

- Bool: *False*
- $[\alpha]$: *[]*
- Map α β: *Data.Map.empty*
- Maybe α: *Nothing*

Any user that wants to define their own type, has to implement an instance of *DefaultValue* type class for the type they want to use as a *Hanus* variable. It is possible to implement a *DefaultValue* for a type without implementing any operators for that type. A variable of this type can not be changed in this case, except if the type also implements the type class of another type. An example of this would be the class *Real*. Because any *Real* in Haskell has to be also a *Num*, all operators defined on *Num* could also be used on a *Real*.

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

3.1.2.2 Definition of operators

In *Hanus*, operators can be defined to manipulate variables. These operators always come in pairs, one operator for the forward manipulation and one operator for the reverse manipulation. There is no restriction that these operators should be different. For example, the reverse of the negation operator is the negation operator. However, in most circumstances, there operator are two different manipulations. In the original *Janus*, there were 4 manipulation operators defined. There are the operators `+=`, `-=`, `^=` and $\Leftrightarrow$. All of these operators only work on integers, as the original *Janus* does not support any other types. The semantics are as follows:

- `x+=e` adds any arbitrary expression `e` to the variable `x`. The reverse of this statement is `x-=e`. As long as the variable on the right hand side of the operator is not used in the expression e, e can by any computation that results in an integer. The expression `x+=x` is therefore not allowed, because this statement does not have a reverse. The constraint that the variable on the left can not occur on the right applies for all operators.

- `x-=e` subtract the arbitrary expression `e` from the variable `x`. The reverse is the statement `x+=e`.

- `x ^= e` does a bitwise *xor* with the expression `e`. This operator is its own reverse.

- $x \Leftrightarrow y$ is the swap manipulator. This operator takes two variables and swaps their values. This operator is its own reverse.

For a user to implement their own operator in *Hanus*, they have to define their operator to be the of the type *Operator*. The type *Operator* is defined in *Haskell* as *Operator* α β = *Operator* $(\alpha \rightarrow \beta \rightarrow \alpha)$ $(\alpha \rightarrow \beta \rightarrow \alpha)$. The first variable is the operator to be used in forward operation, the second variable is the reverse of the first operator. The definition of `+=` in *Hanus* is `(+=) = Operator (+) (-)`. A user can use the function `inverse` to inverse an *Operator*. The definition of `-=` would become `(-=) = inverse (+=)`. Other operators that are defined for the users convenience include `^=`, swap, push and pop on a stack. All these operators are defined in *Haskell*. Any user wanting to implement their own operators have to define these in *Haskell*, as their is no way to define these in *Hanus*.

3.1.2.3 Guarantee of reversibility

The *Operator* type does not guarantee reversibility. This means that there is no check that if a user specifies two manipulations m_1 and m_2 that $(x\ m_1\ e)\ m_2\ e = x$ holds for every variable x and every expression e. It is possible that the user defines `(*=) = Operator (*) (+)`. It should be clear that this operator is incorrect, because the reverse of multiplication is not addition. However, *Hanus* will not detect such a mistake and will run a program containing this code without problem. Therefore, a user that defines their own operator, should be careful to make sure that every operator they define is indeed reversible in every situation.

3.1.3 Indexers

Indexer are functions that are used to access specific fields in a data structure. These data structures are are implemented in *Haskell* using the *DefaultValue* type.

3.1.3.1 Field indexer

A field index is an indexer used to point to a specified field of a data type. One example, implemented in *Hanus*, is the type *BinaryTree*. This *BinaryTree* is either a *Node* with a value and two sub trees or a *Leaf*. The *DefaultValue* of a *BinaryTree* is a *Leaf*. This *BinaryTree* is not very useful unless there would be an option to access the values or children of the tree. This is where *FieldIndexer* is used. To define a *FieldIndexer*, a user should specify both a *get* and a *set* function. The *get* function is used to retrieve the value from the data structure and the *set* function is used insert the value in the data structure, possibly removing the data that was there before. This is not a problem for reversibility, because this *set* and *get* are never directly called by the program at runtime. They are used by the operators, retrieving the value, updating it and returning the updated value using the *set* function. A few examples of the usage of a *Field Indexer* include $x.nodeValue$ and $x.leftChild$. Each of these fields can then be used as a variable with any operator that support the type of the field. The details of the implementation of *BinaryTree* can be found in the code repository in the module $StdLib.BinaryTree$.

3.1.3.2 Array indexer

Array indexer have the same purpose as Field Indexer in that they are used to access specific parts of the data structure. The difference between the two indexers is that an *ArrayIndexer* has one more variable. This variable is used to access a specific part of the data structure that is not necessary known at compile time. The most notable example is the default way access items in an array: $x[i]$ where x is an array and i is the key that points to a position in array x. In some languages, there is a restriction that the key has to be an integer. This restriction does not exists in *Hanus*. A user is free to use any type as a key. For the definition, the *ArrayIndexer* also requires two functions, one *get* function and one *set* function. The *get* function requires the variable that needs to be accessed and a key that points to an item in the data structure. The *set* function requires a function that receives the data structure, the key of the variable in the data structure and the new value that needs to be inserted in the data structure. In *Hanus*, $ArrayIndexer$ is used to implement arrays. This indexer can also be used for structures like maps, lists and similar structures.

3.2 Examples of ***Hanus***

DIVISION

```
{-# LANGUAGE TemplateHaskell, ScopedTypeVariables, QuasiQuotes, FlexibleContexts #-}

module Divide where

import QQ
import StdLib.Operator
import StdLib.DefaultValue

[hanusT|
procedure divide(x :: Int, y :: Int, z :: Int){
    from x >= y && z == 0 loop
        z += 1;
        x -= y;
    until x < y;
}

procedure main(x :: Int, y :: Int, z :: Int){
    call divide x y z;
    uncall divide x y z;
}|]
```

The division example is a simple example of what *Hanus* is capable of. This function computes the division and the remainder. When the variable is called in reverse, it can be used to compute the multiplication. Because the *main* procedure calls divide and then uncalls divide with the same arguments, main acts as the identity function for most parameters. With this program, it is possible to explain the loop in *Hanus*. The loop has a precondition and a postcondition and a body. The precondition has to be *True* only the first iteration and the postcondition has to be *True* only the last iteration. When the loop is reversed, precondition and postcondition change places.

Fibonacci

```
{-# LANGUAGE TemplateHaskell, ScopedTypeVariables, QuasiQuotes, FlexibleContexts #-}

module Fibonacci where

import QQ
import StdLib.Operator
import StdLib.DefaultValue

[hanusT|
procedure fib(x1 :: Int, x2 :: Int, n :: Int){
    if n == 0 then
        x1 += 1;
        x2 += 1;
    else
        n -= 1;
        call fib x1 x2 n;
        x1 += x2;
        swap x1 x2;
    fi x1 == x2;
}

procedure main(x1 :: Int, x2 :: Int, n :: Int){
    call fib x1 x2 n;
}
|]
```

The Fibonacci example has three parameters. The first two are two sequential Fibonacci numbers, the third parameters n represents the nth parameter. This function can normally be used to calculate the nth Fibonacci number. The reverse of this program is, given two sequential Fibonacci numbers, what is the position of the first Fibonacci number when all Fibonacci numbers are listed. For example, $x1 = 5$ and $x2 = 8$ return $n = 4$ because 5 is the fourth Fibonacci number, starting on a list with 0. This example also includes an if-statement. This if-statement differs from regular if-statements with the addition of the postcondition. This postcondition must be $True$ if the if-branch was executed and has to be $False$ if the else-branch was executed. The reverse of the if-statement is the reverse of both bodies and the reversal of the precondition and the postcondition.

Run length encoding

```
{-# LANGUAGE TemplateHaskell, ScopedTypeVariables, QuasiQuotes, FlexibleContexts #-}

module RLE where

import QQ
import StdLib.Operator
import StdLib.DefaultValue

[hanusT|
procedure encode(text :: [Int], arc :: [Int]){
    from (text /= []) && arc == [] do
        local val :: Int = 0;
        local n :: Int = 0;
        val += head text;
        from n == 0 do
            local tmp :: Int = 0;
            pop text tmp;
            delocal val;
            n += 1;
        until text == [] || ((head text) /= val);
        push arc val;
        push arc n;
        delocal 0;
        delocal 0;
    until text == [];
}

procedure main(text :: [Int], arc :: [Int]){
    call encode text arc;
}
|]
```

The run length encoding example shows the usage of a more complicated program that uses two nested loops and a stack to store the data. The program compresses the data using run-length encoding. An example is the array $[12, 12, 12, 13, 13]$, which gets converted to the array $[2, 13, 3, 12]$. This is an example with a very useful reverse. A user has only to provide the encoder for such a compression and by uncalling the encode procedure, providing the compressed data, the original data is returned.

3.3 Achievements

3.4 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