Project Report

Concepts of Program Design

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

Problem

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 **??**.

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

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.

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.

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;
}
```

Methodology

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).

Expected outcome

Planning

Results

General workflow

We followed a meta-programming approach to implement our DSL embedding. The programmer writes his Janus program in the `[|hanus|]` quasi-quoter, which then is parsed and checked for semantic errors. If the program is well-formed, the evaluator generates haskell programs corresponding to the given Janus code. In the following section, we give a detailed description of all these parts.

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.

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.

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.

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.

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

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*.

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* $(\alpha \to \beta \to \alpha)$ $(\alpha \to \beta \to \alpha)$. 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*.

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.

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.

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 $StdLib.BinaryTree$.

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.

Static Guarantees

Syntactic Checking

Our TH approach of embedding requires us to eventually write a parser for Janus, which will run at compile-time, during the first compilation *stage*, where our meta-programs are executed in order to produce the final Haskell programs. Therefore, errors encountered during parsing Janus syntax will be reported at compile-time as well.

As an example, the code fragment of our DSL on the left will result in the error on the right:

Code

```
[hanus|procedure main() {
  local n : Int = 10;
    n += 10;
  delocal 20;
}|]
```

Error

```
Exception when trying to run compile-time code:
  Parsing of Janus code failed in file ....
  First error:
--  Expecting "::" at position LineCol 2 10
```

Semantic Checking (Haskell-side)

By embedding Janus in Haskell, we have inherited powerful static semantics from the host language. Therefore, we do not have to implement a lot of semantic checks ourselves, but rather get them for free! As an example, we do not implement errors for using undefined variables, since the generated Haskell programs will be ill-formed and result in a GHC compile-time error. A more interesting example appears in the case of *anti-quoatation*, where the injected Haskell expression has a type-mismatch and we propagate the error GHC will give in this case, as shown in the example below:

Code

```
[hanus|
init :: Int;
a :: BinaryTree Int;
procedure main() {
    createNode a;
    a.nodeValue += map (+ 1) init;
}|]
```

Error

```
- Could not match expected type Int with actual
    type [Integer]
- In the expression: map (+ 1) i
```

Semantic Checking (Janus-side)

By using *Template Haskell* and *Quasi-Quotation* in our approach, we have achieved additional Janus-specific static guarantees. This is possible via insertion of semantic checks during the first compilation *stage*.

One such check is that there exists at least one main procedure, which is the entry point of execution. A more interesting check is that of proper variable usage on the right-hand side of an assignment operator, namely that it should not contain the right-hand variable being mutated. The following example demonstrates this check:

Code

```
count :: BinaryTree a -> Int

[hanus|
a :: BinaryTree Int;
procedure main() {
    createNode a;
    a.value += (count a.left) + a.value
}
|]
```

Error

```
Semantic Error (line 6, col 23):
    Assigned variable appears on the left-hand side.
```

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 == [];
}
```

```
27
28  procedure main(text :: [Int], arc :: [Int]){
29      call encode text arc;
30  }
31  |]
```

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.

Evaluation

An important part of Hanus is the eval module. Eval is responsible for the actual Hanus compilation: it takes a Hanus AST and compiles it to Haskell functions. In this section, the chosen compilation tactic for some of the more important or interesting types of the Hanus AST will be discussed.

General approach

The general task of the evaluator is to convert a provided Janus program to a collection of top level declarations that can be spliced by the user and run just like regular Haskell code. The main challenge lies in the fact that Janus is an imperative language with mutable global state, while Haskell is neither imperative, nor does it have any sort of global state.

This problem can be solved by observing that a statement in an imperative language can in principle be viewed as a function that, given the current state of the program as its input, yields a new state as the result of executing the statement. All Hanus statements (with the exception of the *log* statement) may only affect the variables that are in scope at that point, so we consider the program state for any statement to be the collection of variables that is in scope when executing that statement.

Procedure declarations

Every Hanus procedure declaration is compiled to a Haskell functions, consisting of a single do-statement, that performs a computation that is equivalent to that intended by the Hanus procedure. Every Hanus statement in a procedure is converted to one, or sometimes more than one statement in the Haskell do block. Since a procedure can be called to run it regularly and uncalled to run it in reverse, every Hanus procedure p is compiled to not one but two Haskell functions: a function p that does exactly what Hanus procedure p should do, and a function p' that is the inverse of p. Name conflicts as a result of this convention are impossible since the Hanus parser does not accept the $'$ character as part of a procedure identifier.

Global variables

Hanus has two types of variables. Global variables are declared at the top level with a name and type. The programmer cannot specify a default value for the variable, but instead the variable will automatically get the default value of the specified type. Since global variables can be updated from anywhere in a Hanus program, Hanus imposes the restriction on the programmer that global variables cannot be passed to procedures. Furthermore, it would seem to be a sensible choice to substitute *IORefs* for global variables since *IORefs* are a Haskell solution to variables. However, the problem would then be that the entire Hanus AST and possible Haskell sub-AST's (since any Haskell expression can be written on the right hand side of a Hanus expression) would have to be traversed, replacing all variable declarations and expressions with *readIORef* and *writeIORef* operations, which would have been quite a challenge. Instead, global variables are simply declared as lets. If they need to be updated, they are just redeclared. Note that something like `a += 1` is not compiled to `let a = a + 1`, because this causes an endless loop due to `a` being on the left and right hand side. Instead, it is compiled to `let temp = a; let a = temp + 1`. To handle all difficult cases where there are function calls or nested statement blocks that (possibly) shadow the let declaration, but only in the nested scope, a simple call-by-copy-restore mechanism is used: the Haskell counterpart of any Hanus procedure takes all global variables as inputs, and the last statement in the do-block of the Haskell function returns all global variables in a single tuple. The caller of the procedure then redeclares all global variables after the procedure call. An example:

```
a :: Int;
b :: Int;
procedure x() {
    call y;
}
procedure y() {
    a += 1;
    b += 1;
}
```

This procedure is compiled to something like the following:

```
x a b = do
    let temp = y a b
    let (a,b) = temp
    (a,b)
y a b = do
    let temp = a
    let a = temp + 1
    let temp = b
    let b = temp + 1
    (a, b)
```

Local variables

The other type of variable is the local variable. Local variables are declared in a *local* statement with a name, type and default value, and undeclared in an *unlocal* statement with a final value. By the time the unlocal statement is being executed, the local variable should have the specified final value. Note however, that the final value can also be a non-literal expression. The unlocal block is there to function as initial value for the local variable if the procedure that it is part of is being uncalled, i.e. reversed: in that case, the local variable needs what is declared as its final value as its initial value, and vice versa. By the time the unlocal block is reached, an assertion is executed to make sure that the procedure can safely be reverted without giving the local variable an incorrect initial value. An unlocal block is implemented as a `let-in` Haskell statement, because this automatically takes care of removing a local variable from scope after the unlocal line. To keep it simple, nesting multiple local blocks with identical variable names is impossible, and local variables can never have the name of a global variable. The let-in block again causes scoping issues, for example here:

```
a :: Int;
procedure x() {
    local b :: Int = 10;
    a += b;
    unlocal 10;
```

Here, `a` will be redeclared in the `let-in` block, but as soon as the block ends, it will get its old value of `10` again.Therefore, local blocks too make use of the restore functionality that procedures make use of, making the final generated Haskell code look roughly like this:

```
x a = do
    let temp = let b = 10 in do
        let temp2 = a
        let a = temp2 + b
        if b == 10 then
            (a)
        else
            error <message>
    let (a) = temp
    (a)
```

Local variables can be passed around to functions, since they are handled as reference types. An example of a procedure that takes local variables is `procedure x(n :: Int)`. In order to be able to correctly update local variables, they too were made part of the copy restore system, meaning that that the Haskell equivalent of procedure `x` takes all global arguments as input, plus the local variables `n`, and that the Haskell function returns them all in a single tuple. Since local variables cannot shadow global variables, and since global variables cannot be explicitly passed to procedures, duplicate naming issues will never occur.

Log statement

A special type of Hanus statement is the **#log** statement. It uses the `Debug.Trace` Haskell module to print the current value of the specified variables. However, Haskell is a lazy language, so a sequence of Hanus statements like `#log a; a += 1;#log a;` will write "1" and then "0" to the console. Therefore, `#log` statements are strictly evaluated using a bang pattern, making the Haskell code look something like this:

```
let !irrelevant = trace a 0
let temp = a
let a = temp + 1
let !irrelevant = trace a 0
```

If the user has not imported `Debug.Trace` but *does* try to use the `#log` statement, the Eval module will throw an error, saying that they need to import `Debug.Trace`. A problem with logging is that it can cause a change in behaviour of the Hanus program in rare cases, for example if the user tries to log a variable that has a value of 1/0 but that is not used anywhere. Without the logging, it would be completely ignored by Haskell, which is a nice feature. However, as soon as the user starts to log the variable, the program will start to throw errors. This could be solved by using the `-XStrict` language extension, forcing the entire Hanus program to be evaluated strictly.

If statements

The conversion of `if` statements to an equivalent Haskell representation is rather straightforward, since Haskell includes a built-in `if then else` construct. The conversion can be done by simply generating `do` blocks for both branches, wrapping them in an `if then else` construct, and updating the program state with the resulting expression. Since the guard in Janus is in fact already a Haskell expression, it can be transferred directly to the generated `if then else` expression.

Loop statements

Compiling the *loop-until statement* to Haskell was a particularly challenging task. A single while-iteration is simple enough: just execute the body of the loop and use the by now well known restore mechanism to update values in the outer scope. Two loops is simple too: just nest the do-blocks and let them all return the restore tuple on the last line. Same goes for n loops. However, a challenge arises as soon as n is unknown at compile time, because how should the compiler know how many loop bodies it should nest? The obvious solution to this problem is to compile a loop a recursive function that uses the copy-restore mechanism and keeps calling itself at runtime until its guard evaluates to false. The actual loop statement can then be replaced with a simple call to the loop function. However, different loops can be placed in different procedures that can have different local variables, so the input arguments for the Haskell loop function needs to depend on the particular while loop and its scope. This problem can be solved by generating a unique loop function for every loop declaration. However, the problem is that, at the time the loop was being implemented, no scope log was actively being kept during the code generation process, meaning that the Eval module could not actually

know what the signature of the recursive loop function had to become, nor could it know how it could correctly call that Haskell function. Therefore, it was necessary to keep track of the scope everywhere in the Eval module, meaning that every time a local variable was being declared or undeclared, the scope had to be updated. By doing this, whenever Eval encountered a loop block in the AST, it knew exactly what the function signature had to be of the loop function it had to generate, what the type of the return tuple of that function had to be, and what the function call had to look like that had to function as substitute for the original loop block.

Entry point generation

For convenience of the end user, the provided QuasiQuoters provide an extra generated function `run` that calls the generated `main` function. Simply calling the `main` function itself would also be perfectly possible, but since it is actually a function with type `State -> State`, the programmer would have to provide an initial state for the program. Since all variables in Hanus are assumed to have a defined default value, this initial state is implicit. The provided entry point generates this initial state from the types of all global variables and calls the `main` function.

Array and field indexers

Hanus allows for the user to define field indexers for any data type or to make any data type indexable. Running the Hanus program below would result in a value of `(4,2)` for `t` and `["foo" => 1, "bar" => 2]` for `a`.

```
t :: (Int, Int);
a :: Array String Int;

procedure x()
{
    t.first += 2;
    t.second += 4;
    swap t.first t.second;
}

procedure y()
{
    a["foo"] += 1;
    a["bar"] += 2;
}

procedure main()
{
    call x();
    call y();
}
```

Due to the way these functionalities are implemented at the Haskell level, it is necessary to use a slightly different restoration strategy when assigning a value through a field or array indexer. This is a result of the fact that applying assignment operators to field or array indexes does not yield a new value for the actual data structure, but rather a new value for a certain field or index within that structure. The structure itself still has to be updated with the new value after applying the operator. To solve this problem, all variables that are affected by an operation are restored individually instead of all at once. Compare the restore mechanism demonstrated in section on *global variables* with the code that is generated for the example shown above:

```
-- Similar to as demonstrated in the section on global variables
let temp = swap (a,b) ()
let (a,b) = temp

-- Partial (simplified) splice generated for the procedure
-- x() in the example above
let (v1, v2) = swap ((get first) t), ((get second) t)) ()
let tmp1 = (set first) t v1
let t = tmp1
let tmp2 = (set second) t v2
let t = tmp2
```

Note that the shown splice in the second example is a simplified version of the actual splice, `get` and `set` represent functions that extract the necessary function from the predefined field indexers `first` and `second`.

Unfortunately, nested field or array indexers are not yet supported due to limited time being available. In order to support nested indexers the restore mechanism would have to be expanded in order to cope with updating nested structures. In general, the generated code for an assignment such as `t.first.first += 1;` would look something like this:

```
let v = (+=) ((get first) ((get first) t)) 1
let tmp1 =
    let tmp2 = (set first) ((get first) t) v
        in (set first) t tmp2
let t = tmp1
```

Naming

In order to create a functioning program, the evaluator generates quite a lot of helper variables and functions. Names provided by the input program are kept the same (GHC will resolve them for us). In all other cases, Template Haskell provides the function `newName` that generates a name that is guaranteed to be unique inside the current scope.

The usage of this method was not always free of problems unfortunately, since GHC would consider the names of generated helper functions to be out of scope in some cases, even though their declaration would be spliced at top level. The exact source of this

problem remains unclear, but we were able to circumvent this issue by moving the declaration of any helper functions to a where clause attached to the function in which they are needed.

Runtime errors

Hanus programs can throw errors at runtime that cannot be predicted at compile time without evaluating the actual Haskell program, e.g. checking if the local variable has the value specified in the delocal block by the time that delocal block is executed. Therefore, the Eval module embeds errors in the Haskell program structure. These error messages are very detailed in certain cases, to make it easier for the programmer to find out what is going on and where. After all, all information about line numbers or exact structure is lost during compilation. All in all, the previously mentioned example of checking the delocal block will roughly be compiled to the following (note that this is code is simplified):

```
if x /= expected_value then
    error "variable x has value '" ++ (show x) ++ "' although it should " ++
          "have had the value '" ++ (show expected_value) ++ "'."
else
    continue
```

Benchmarking

To compare our Janus implementation with the extended version currently available (i.e. *Jana*), we used the *criterion* package, which benchmarks a given Haskell program by executing it multiple times and, using statistical methods, derives an accurate estimate of its execution time.

We benchmarked the aforementioned three examples, namely division, fibonacci and run length encoding. For each one of them, we provide measurements for small and large inputs. We also measure a "tweaked" version of the program which, instead of just calling the `main` procedure, loops 1000 times and always uncalls `main` immediately after calling it. This nicely demonstrates the performance we gain by generating the reverse procedure at compile-time.

	division_{small}	division_{large}	division_{loop}
jana	72μs	81.06μs	17.16ms
hanus	22.57μs	28.18μs	21.08ms

Table 1: Division benchmarks

	fib_{small}	fib_{large}	fib_{loop}
jana	80.5μs	4,6ms	57.73ms
hanus	16.9μs	253.5μs	21.5ms

Table 2: Fibonacci benchmarks

	rle_{small}	rle_{large}	rle_{loop}
jana	151.1μs	1.01ms	1.1s
hanus	24.4μs	26.56μs	135.3ms

Table 3: Run-Length-Encoding benchmarks

From the measurements above, it is apparent that *Hanus* outperforms *Jana* in both small and large inputs. In the case of the "tweaked" benchmark setup, if the input program is large enough, we have very large performance gains. This is due to the fact, that *Jana* constructs the reverse program everytime an `uncall` is evaluated, leading to huge runtime overhead.

Last but not least, we have not included compilation time, which is expected to be a lot more in *Hanus*, due to extensive amount of computation happening at compile-time. In favor of accuracy, we have not counted the correspondent runtime overhead that *Jana* introduces for parsing and semantic checking.

Goal/planning adjustments

Reflection

Good/bad surprises

As was already stated in section 3.5, it was very challenging to implement while loops. Part of it was the difficult evaluation structure of a Janus / Hanus loop (reference to original Janus paper), but another large aspect was the fact that it was all of the sudden necessary to keep track of the current scope *everywhere*. Since all basic Hanus features were being implemented in Eval before the more advanced statements were being added, a tactic that was not bad per se, the risk always is that you have to rewrite code because an advanced feature does not play nice. This was the case here: a substantial amount of Eval code had to be adapted or rewritten and this took up quite a bit of time. Before Eval was rewritten, a not unsubstantial amount of time was also put into trying to find a way to make inlining a loop possible using higher order monad functions, but this research sadly enough did not pay off.

Problems along the way

Although everybody was aware of the amount of work that had to be put into the Eval module from the get go, it still cost a lot more time eventually. For a problem that, we all already recognised back then, was going to be difficult, more time should have been claimed since a problem that seems theoretically challenging is almost always far more difficult in practice. However, a mistake was also made by not correctly working in a strictly iterative fashion. The result of this was that the Eval module was almost always broken in one way or another because it was still not finished. Working in a strictly iterative fashion, always working on one function and one function only, would have resulted in a clearer overview of what the tasks and goals were at every moment,

and in clearer moments on which the only partially finished, but stable, Eval module could be merged with the master branch.

Due to an initial lack of tests for all possible Hanus constructs, progress on the evaluator was slightly delayed towards the end of the project, since a few previously unknown bugs in the parser became exposed at that time.

Future work

The Hanus project can be extended or improved in multiple ways. Some ideas for future work on Hanus are:

1. Implementing proper field and array indexers, i.e. at the moment, a sequence of multiple field and array indexers is not supported

2. Better error messages that don't look like Haskell errors. This is not a bug per se, but when a programmer writes complicated Hanus code, they will quickly find out that the error messages are very difficult to debug, partially because the error messages concern the generated Haskell code, but more so because of the fact that the structure of certain Hanus expressions such as loops is rigorously changed in the compilation step to Haskell, making it very hard for the programmer to find out what is wrong in the actual Hanus program.

3. Template Haskell bug fixes. During the development of Hanus, multiple Template Haskell bugs have been found that are most probably caused by Template Haskell, e.g. function declarations that Haskell cannot find when they are declared at the top level, but that Haskell *can* find when they are defined in a where clause of the callee, or variables that should get a unique name from Template Haskell, but still cause "multiple declarations of x" errors in Haskell.

4. Fixing the endless loop bug in the parser. Although the error correcting parser works great, it can sometimes get stuck in an endless loop if the programmer writes a program that has incorrect statements.

Appendix

Repo link

The public repository can be found at `https://github.com/joristt/Hanus`.

Code navigation

- `Hanus`
  - `proposal`: Source code for initial proposal
  - `intermediate`: Source code for intermediate results
  - `report`: Source code for this report
  - `poster`: Source code for poster presentation
  - `presentation`: Source code for Janus lecture
  - `janus`:
    - `app`: Executable's entry point
    - `examples`: Example programs
    - `src`
      - `Parser`: Parsers for Janus and Haskell
      - `StdLib`: Hanus' Prelude
      - `AST.hs`: Defines Hanus' abstract syntax tree
      - `Eval.hs`: Code generator from ASTs
      - `QQ.hs`: Quasi-quoter
      - `ReverseAST.hs`: AST reversal
      - `SemanticChecker.hs`: Semantic checks
    - `test`: test suite
    - `janus.cabal`: Cabal file