$\mathbb{J}_<$: A Time Reversible Programming Language

CPL: Project assignment 2023 - 2024
Denis Carnier
denis.carnier@kuleuven.be
Version 1, dated November 18, 2023

Contents

1 Introduction

All software implementations contain bugs. This project is no different. If you notice (what you believe to be) a bug or mistake in the starter files, please signal it to me so I can take appropriate action.

Most software *specifications* also contain bugs. This document is no different. If you notice (what you believe to be) a bug or mistake in this assignment, please signal it to me so I can take appropriate action.

In this project, you will use OCaml as a metalanguage to implement a simple, time-reversible programming language called $\mathbb{J}_<$ (pronounced "Jay sub"). A time-reversible programming language is one that in addition to the usual interpretation of computation as calculating a program's output from some input, also allows computing a program's inputs from its output. Even if you have not seen an explicitly time reversible programming language before, you will probably have seen some of its applications. Examples include reversible debuggers [2, 7] with operations to "step forward" and equally to "step backward"; optimized procedures in cryptography [5, 8] and graphics [1]; and quantum computers [6] that operate over a set of (universal) reversible gates that disallow the loss of information (and the increase in energy dissipation, known as Landauer's principle [3]) typically associated with a non-reversible computation. For this project assignment, we will borrow heavily from the design of the Janus [4] language, but implement only a limited subset. Concretely, your implementation of $\mathbb{J}_<$ will support at least the following (mandatory) language constructs:

- declaration of global program variables;
- imperative assignment through so-called "swap" and "modification" statements;
- conditionals through augmented `if`-statements;
- a looping construct;
- and procedure definitions with procedure calls and "uncalls".

These language constructs will be supported by the following procedures over $\mathbb{J}_<$ programs (that you should implement):

- a typical forward evaluator `feval` (such as the ones seen in the EOPL textbook);
- a backward evaluator `beval` that "uncomputes";
- a program inverter `invert`;
- and possibly (as part of the optional assignments) an optimizing source-to-source compiler `optimize`.

Reversibility of statements To give some additional context, we can interpret forwards and backwards evaluation as two denotations $[\![\_]\!]_F, [\![\_]\!]_B : \text{Proc} \to \text{Store} \to \text{Store}$, respectively. For all $\mathbb{J}_<$ procedures p it holds that

$$\forall(S \in \text{Store}).[\![p]\!]_F(S) = S' \implies [\![p]\!]_B(S') = S. \tag{1}$$

Said differently, If forward evaluation of a procedure p over an input store S results in an updated store S', then backward evaluation of p over the updated store S' will "uncompute" to the original store S.

What is expected? We expect you to spend up to **40 hours** on this assignment. The project consists of a number of mandatory assignments and a number of optional assignments. Each assignment requires you to implement a particular language extension for $\mathbb{J}_<$. You are **not required** to complete the optional assignments. However, the more optional assignments you complete, the higher your grade will be. Please refer to §2.4 for a breakdown of how your grade is calculated.

The remainder of this document outlines in greater detail the rules and practicalities that you **must** abide by (§2); what you have to implement (§3); and what you may implement for a higher grade (§4). We finish with some small $\mathbb{J}_<$ example programs (§A).

2 Practical matters

2.1 How to get started

You are now a maintainer of a KU Leuven GitLab (`https://gitlab.kuleuven.be`) repository titled `jaysub-XXXXXX` where `XXXXXX` is your student number. If this is not the case, it is likely because you have never logged into KU Leuven GitLab (and thus don't yet have an account). Please ensure that you have at least authenticated with the KU Leuven GitLab once using your credentials, then send me an email or pass by my office (200A 5.34).

To help you hit the ground running, this repository contains a starter template. This starter depends on a number of third party OCaml packages. Begin by installing them on your system. Refer to the OCaml exercises if you have problems.

```
opam install dune menhir ppx_deriving
```

The starter defines an abstract data type of $\mathbb{J}_<$ programs (`Ast.program`, under `lib/ast.ml`). And a parser and lexer for a surface syntax of $\mathbb{J}_<$ that is used in the example programs in §A. They are located under `lib/lexer.mll` and `lib/parser.mly`, respectively. When you run `dune build`, these files will be compiled into valid OCaml modules.

Implement `S` Your task is to implement the module `Solution`, found in `lib/solution.ml`. Note: you should not update the module signature `lib/solution.mli`. It is sufficient that your solution **implements** this interface. You will notice a number of `failwith`s in the current implementation. Replace these at appropriate times (i.e. when a green box in this assignment tells you to) with your implementation.

TDD To improve your implementation, we recommend working in a test-driven development style. Widely-used testing frameworks include OUnit2 and Alcotest, but you can also implement your own. TDD follows a simple recipe: design a test case, then write your implementation until the test passes (`dune test`). Rinse and repeat until you run out of ideas for new test cases and all tests pass.

Running To try your code on example programs, you can make use of the command line interface (CLI) that is defined in `bin/main.ml`. You should not need to edit this CLI. Simply invoke it using e.g. `dune exec jaysub forward examples/fib.jsub` to try

it out. Don't forget to implement `string_of_store` in the `Solution` module so the CLI can print the final store.

Be creative! Coming up with good test cases is important to getting a correct implementation. Conceive unit tests that test specific functions, as well as integration tests: write new $\mathbb{J}_<$ program (besides the ones in the `examples/` directory), run them, and compare the output with what you expected.

Read this document in its entirety Do not wait until handing to read the appropriate section! Read the rest of this document now. It contains useful information!

2.2 Deadline

The deadline for this project is **Friday December 22, 2023, at 23:59:59 CEST**[1].

2.3 Handing in

Before submitting, confirm that you have answered all questions (and that your answers are up-to-date) in the `ANSWERME.md` file, which can be found in the project's root directory.

To submit your project, it is sufficient that you push your solution to the `main` branch of your automatically generated personal GitLab repository for the project. When the clock strikes midnight on December 23, your repository will be archived, meaning you will lose write access to any of the branches. The last commit on `main` will act as your submission.

A free tip Work in a local branch that is different from the `main` branch. Periodically, for instance after you have completed a particular assignment, you can fast-forward the main branch to bring it up-to-date with your local branch. This ensures that the main branch always contains an implementation that can be **successfully compiled**.

Another free tip Schedule ample time *before* the deadline to push your code and polish your submission. Attempting this for the first time minutes before the deadline is a sure way to mess something up and submit a solution that does not conform to the grading requirements outlined below.

2.4 Grading

N.B. There are **no project retakes** in the second session. Your project's grade from the first session is automatically transferred to your retake of the written exam in the second session.

Correctly implementing all mandatory assignments (everything in §3) nets you 3 points, out of a maximum of 5 points, provided that the following requirements are also met:

[1] ISO-8601: `2023-12-22T22:59:59.000Z`. UNIX epoch: `1703285999`.

- Your submission does not produce any compilation errors using a recent ($\geq$ 4.14.0) version of the OCaml compiler.
- Your submission is extensively[2] documented using `ocamldoc` documentation comments and style.
- Your submission is thoroughly tested, both at a unit level, and at an integration/end-to-end level.
- Your submission is written (to the best of your ability) in idiomatic OCaml.
- You have filled in the `ANSWERME.md` file in the project root.

What does *correctly implemented* mean? The intended behaviour of the functionality is apparent in your implementation. This behaviour includes the normal behaviour and edge cases of the functionality in isolation, as well as behaviour that arises from the interplay with all other language features/assignments that you submit for grading.

Beyond a passing grade

Implementing any of the optional assignments from §4 in a correct manner will net you one (1) bonus point over the base grade you received for the mandatory part of the project, up to a maximum of 5 total points for this project. As stated before, you are free to implement zero or more of the optional assignments, in any order. Note: the same additional requirements of the original assignments (does compile, is documented, is tested, is idiomatic) apply.

Defending your project

You may be asked to present and defend your project orally some time during the first session examination session. More information will be communicated closer to that date.

2.5 Need help?

If you have a question about the project or found a bug in the code, post it on the discussion forum on Toledo. This forum is not a helpdesk for OCaml basics. Questions that can be answered through a simple web search will not be answered. Keep the rules about plagiarism in mind when posting, so don't share code, use abstract examples.

2.6 Plagiarism

You are welcome to discuss your work with other students and with the TA. BUT: The code that you turn in **must be written by you, and by you only**. NO copying or sharing code with others. NO making code available to others. NO using code made available by others. NO large (or small) language models, automatic programming aids, GitHub Copilot, chatGPT, etc. You have been warned!

[2]But not needlessly: write documentation for things that are not evident; refrain from stating the obvious.

3 Mandatory assignments

Implementing all assignments described in this section correctly (see §2.4), guarantees at least a passing grade (3/5) for the project.

Grammar of $\mathbb{J}_<$ In this overview, we give only an informal grammar of $\mathbb{J}_<$ programs Prog as an implementation aid. For a complete, formal grammar and semantics of the Janus language, see [9]. Note that in the starter project, we provide you with a lexer and parser for a surface syntax of this grammar. The parser produces a term of type `Ast.program` that you must use as the representation of $\mathbb{J}_<$ programs.

Prog	::=	Decl*; Proc+
Decl	::=	**id**
Proc	::=	`procedure` **id** { Stat }
Stat	::=	Stat ; Stat
	|	**id** Modi Expr
	|	**id** $\Leftrightarrow$ **id**
	|	`if` Cond `then` { Stat } `else` { Stat } `fi` Cond
	|	`call` **id**
	|	`uncall` **id**
	|	`skip`
Modi	::=	$+= \mid -= \mid *= \mid /=$
Expr	::=	**num** | **id** | Expr + Expr | Expr − Expr | Expr ∗ Expr | Expr/Expr
Cond	::=	Expr `==` Expr | Expr `!=` Expr | Expr

Programs Prog consist of zero or more variable declarations Decl, and one or more procedures Proc. Executing a $\mathbb{J}_<$ program corresponds to calling the last defined procedure. By convention, the example programs in Apdx. A will name this final procedure "`main`", but this is not a requirement in the grammar of $\mathbb{J}_<$. A declaration Decl is simply a variable identifier. In the surface syntax, this corresponds to a string. Procedures Proc are defined using the `procedure` keyword, followed by a name and a statement Stat enclosed in curly braces. This statement corresponds to the procedure body. The remaining productions are discussed in the following sections.

3.1 Basic control flow

The first mandatory assignment is to implement a core set of language primitives and an evaluation function `feval`. Concretely, you will implement reversible imperative assignments and conditionals, and an evaluation function that given a $\mathbb{J}_<$ program containing some global variables (initialized to 0) and at least one procedure, evaluates the body of the last procedure. The result of `feval` is a final variable store: a container that maps each of the declared program variables at the beginning of the program to a final output value.

Global variables A $\mathbb{J}_<$ program consists of a list of a global variable names and a program body. These global names point to integer values and are by convention initialized to zero (0) (see some example in Apdx. A for what this looks like). The program may only read and write to those variables explicitly declared at the beginning of the input program. Otherwise, an error should be raised.

Modification statements A global variable can be written to in $\mathbb{J}_<$ in two ways. The first is using a modification statement:

```
x += 2;      y -= 5;      z *= x;      a += x + y;      q *= x + y - z;
```

This statement mutates a declared variable by first evaluating the right-hand side (RHS) expression, and then overwrites the value of left-hand side (LHS) program variable by an application of the binary operator to the original value of LHS and the result of the RHS expression. N.B. To ensure that these statements remain reversible, there is a syntactic restriction that the LHS variable does not appear in the RHS expression.

Swap statements The second way to mutate variables is by swapping values. Two program variables x and y may swap their associated values at any point during the program by invoking the swap statement `<=>`:

```
procedure ... {
  ...
  // (A)
  x <=> y;
  // (B)
  ...
}
```

At point (A) x holds the value p, and y holds q. After executing the swap, at point (B) x now holds q, and y holds p.

Conditional statements Conditional statements are modelled using a modified `if-then-else` construction. As opposed to their traditional counterparts, a reversible conditional has two predicates: a classical *test* at the start (between the `if` and `then` keywords) and a final *assertion* at the end (after the `fi` keyword). If the *test* evaluates to a truthy value (N.B. all integers except zero are considered truthy), the `then`-branch is executed with the requirement that afterwards the *assertion* must hold (if it does not hold, the conditional is undefined and execution should halt with an error). Similarly, if the *test* is false, the *assertion* must be false after executing the `else`-branch.
Examples of conditionals can be found in the example programs (see §A).

Assignment Implement `feval` in the solution module. Ensure that programs with one procedure containing modifier statements, swaps and conditionals execute correctly.

3.2 Procedures

This next mandatory assignment requires you to implement forward evaluation of procedure calls. Recall that procedures do not take any formal parameters. Instead, they manipulate the program's global variables which are defined at the start of $\mathbb{J}_<$ source programs. A procedure call `call PROCNAME` executes `PROCNAME`'s corresponding procedure body with the global variable store. If `PROCNAME` does not correspond to a valid procedure name, then behaviour is undefined.

Besides procedure calls, $\mathbb{J}_<$ also includes procedure *uncalls*. Uncalls invoke the inverse computation of the procedure, and as such, can be seen as a regular procedure `call` of the inverse procedure body. Program inversion is discussed in §3.5.

Assignment Extend your forward evaluator from the previous assignment with support for procedure calls and uncalls.

3.3 Loops

In this mandatory extension, you will augment the forward evaluator with a looping construct. Here is an example of loop syntax:

```
procedure ... {
  ...
  from (x == 0) do {
    y *= 2;
    x += 1;
  } until (x == 10)
  ...
}
```

A reversible loop has two predicates (not unlike the conditional): the predicate after `from` is the *assertion*, and the predicate after `until` is the *test*. Initially, (a) the assertion `x == 0` must hold, and then (b) the loop body `y *= 2; x += 1;` is executed. After one iteration of the loop body, (c) the test `x == 10` is evaluated. If it holds, the loop terminates. Otherwise, the assertion `x == 0` must no longer hold, and we repeat evaluation from step (b) onward. Notice that the loop is reversible by virtue of the assertion, which only holds initially.

Assignment Extend your forward evaluator from the previous assignments with support for `from-do-until` loops.

3.4 Backwards evaluation

Up until now, you have been implementing the forward evaluator `feval` for a language that essentially just has some odd rules for evaluation. We will now define a second evaluator `beval` that implements precisely the backwards evaluation of `feval`. Recall from the reversibility of statements (1) that for a given $\mathbb{J}_<$ procedure p, the following equalities should hold:

$$\begin{aligned} \texttt{feval}(\texttt{call}\ p) &= \texttt{beval}(\texttt{uncall}\ p) \\ \texttt{beval}(\texttt{call}\ p) &= \texttt{feval}(\texttt{uncall}\ p) \end{aligned}$$

Assignment Implement `beval` in the solution module with support for all the language constructs from the previous assignments.

3.5 Program inversion

The last of the mandatory assignments involves the development of a program inverter `invert`. You may view invert as a kind of compiler: it takes a source program p and produces exactly the program p^{-1} that satisfies the following equality:

$$\texttt{feval}\ p = (\texttt{beval} \circ \texttt{invert})\ p = \texttt{beval}\ p^{-1}$$

Theorem (Statement inverter) For all $\mathbb{J}_<$ statements s, the following fact about `invert` ($\mathcal{I}$) holds:

$$\forall(S \in \text{Store}).[\![s]\!]_F(S) = S' \iff [\![\mathcal{I}(s)]\!]_F(S') = S. \tag{2}$$

In words: when forward evaluating a statement s in an initial store S yields an updated store S', then forward evaluating the inverse of s in the updated store returns the original.

Assignment Implement `invert` in the solution module. Concretely, for any $\mathbb{J}_<$ input program you should return a $\mathbb{J}_<$ program that is exactly its inverse.

4 Optional assignments

The optional assignments for this project are largely about an optimizing source-to-source compiler for $\mathbb{J}_<$. In an optimizing compiler, the source program is fed through a number of compiler passes that transform the program in some beneficial way, for instance improving runtime performance, reducing memory usage, or instrumenting the program with additional information used by later passes. Once a number of optimizing passes are defined, a common technique is to apply these passes iteratively: the program is fed through a round of passes, and the resulting program is fed into the same round of passes again. After a number of rounds, or alternatively when the input program remains unchanged after a pass — the compiler's optimization pipeline reaches what is known as a *fixed point* — the compiler returns the result of the final iteration of the pipeline.

Each successfully implemented compiler pass will net you one (1) bonus point, added to your base score for the mandatory assignments. N.B. You **must** complete all mandatory assignments as a requirement to receive a higher grade by implementing any of the following optimizations. It is not necessary that you implement optional assignments in the order in which they are specified here. You are free to tackle each one independent of your progress on any of the other passes.

Assignment Implement `optimize` in the solution module. Concretely, for any $\mathbb{J}_<$ input program you should return an equivalent $\mathbb{J}_<$ program with the optimizations applied that you implemented

4.1 Constant folding and peephole optimizations

Most modern optimizing compilers perform a kind of constant folding: expressions that be fully evaluated at compile time are replaced by their result. This improves the running time of the program, since fewer expressions need to be evaluated. Consider the following code fragment:

```
procedure ... {
  ...
  from (x == y - y) do {
    ...
    x += 1;
    ...
  } until (x == 50 * 20 + 12)
  ...
}
```

This particular example can benefit from constant folding to speed up evaluation of the loop test. This is particularly useful since loop tests are executed after every iteration of the loop body. In the example, the compiler can statically determine the result of evaluating `50 * 20 + 12` and instead produce the optimized test `(x == 1012)`.

Moreover, the loop assertion `(x == y - y)` can be optimized to just `(x == 0)` due to the algebraic laws of subtraction (The same holds for e.g. `(x == y * 0)`). In the literature, this type of optimization is known as a "peephole optimization" that can simplify expressions by for instance algebraic laws.

Assignment Implement `constant_fold` in the solution module's `optimize` definition.

4.2 Dead code elimination

Another optimization pervasive in modern compiler toolchains is *dead code elimination*. The program is scanned for code that is not reachable, and subsequently will never be executed. Consider the following:

```
procedure ... {
  ...
  if (1 != 3) then {
    x *= 0;
    skip;
    skip;
    x += 5;
  } else {
    n += 50;
    skip;
    skip;
    skip;
    call fib;
    x *= 0;
  } fi (x == 5)
  ...
}
```

At first glance, the `skip` statements can all be removed. After all, semantically, `skip` does not mutate the global variable store, and so by definition is dead code. However,

upon closer inspection, you might notice that the condition `(1 != 3)` can be statically reduced to true — after all, the integer constant 1 is never the same as the constant 3. Consequently, the `else` branch will never be executed. We can therefore simplify the entire example to:

```
procedure ... {
  ...
  x *= 0;
  x += 5;
  ...
}
```

N.B. The combination of constant folding and dead code elimination can produce very optimized code! We can imagine a scenario where the condition test was something more complicated like `(4 + 1 == y - y)`, which by constant folding and algebra first reduces to `(5 == 0)` and finally to `0` (false). In turn, the body of the consequent (i.e. the `then` branch) is a candidate for dead code elimination since it can never be reached.

Assignment Implement `dead_code_elim` in the solution module's `optimize` definition.

4.3 Procedure inlining and removing uncalls

Often times, it can be beneficial to reduce the amount of procedure calls. One easy way to do this is through inlining. Essentially, a procedure call is replaced by the body of the procedure being called. Compare the code in the before (left) and after (right):

```
procedure update_vars {
  x += 5;
  y += 3;
}

procedure main {
  ...
  call update_vars;
  ...
}
```

```
procedure update_vars {
  x += 5;
  y += 3;
}

procedure main {
  ...
  x += 5;
  y += 3;
  ...
}
```

In combination with dead code elimination, it might even be the case that the procedure `update_vars` can be safely removed after inlining!

Removing uncalls Due to reversibility, we can translate procedure uncalls of the form `uncall PROCNAME` to statements `call PROCNAME-INV`, where `PROCNAME-INV` is the result of applying `invert` to the body of `PROCNAME`. Use this knowledge to additionally inline procedure **uncalls**.

Assignment Implement `proc_inline` in the solution module's `optimize` definition.

A Example $\mathbb{J}_<$ programs

Fibonacci

```
n, x1, x2;

procedure fib {
    if (n == 0) then {
        x1 += 1;
        x2 += 1;
    } else {
        n -= 1;
        call fib;
        x1 += x2;
        x1 <=> x2;
    } fi (x1 == x2)
}

procedure unmain {
    x1 += 5;
    x2 += 8;
    uncall fib;
}

procedure main {
    n += 4;
    call fib;
}
```

Raising to a power

```
base, exp, res;

procedure raise {
    res += base;
    from (exp != 0) do {
        res *= base;
        exp -= 1;
    } until (exp == 0)
}

procedure unmain {
    res += 1000;
    uncall raise;
}

procedure main {
    base += 10;
    exp  += 3;
    call raise;
}
```

References

[1] Rolf Drechsler and Robert Wille. "From Truth Tables to Programming Languages: Progress in the Design of Reversible Circuits". In: *2011 41st IEEE International Symposium on Multiple-Valued Logic*. 2011, pp. 78–85. DOI: 10.1109/ISMVL.2011.40.

[2] Jakob Engblom. "A review of reverse debugging". In: *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. 2012, pp. 1–6.

[3] R. Landauer. "Irreversibility and Heat Generation in the Computing Process". In: *IBM Journal of Research and Development* 5.3 (1961), pp. 183–191. DOI: 10.1147/rd.53.0183.

[4] Christopher Lutz. "Janus: a time-reversible language". *Letter to R. Landauer*. 1986.

[5] Bikromadittya Mondal, Kushal Dey, and Susanta Chakraborty. "An efficient reversible cryptographic circuit design". In: *2016 20th International Symposium on VLSI Design and Test (VDAT)*. 2016, pp. 1–6. DOI: 10.1109/ISVDAT.2016.8064874.

[6] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[7] Robert O'Callahan et al. *Engineering Record And Replay For Deployability: Extended Technical Report*. 2017. arXiv: 1705.05937 [cs.PL].

[8] Himanshu Thapliyal and Mark Zwolinski. "Reversible Logic to Cryptographic Hardware: A New Paradigm". In: *2006 49th IEEE International Midwest Symposium on Circuits and Systems*. Vol. 1. 2006, pp. 342–346. DOI: 10.1109/MWSCAS.2006.382067.

[9] Tetsuo Yokoyama and Robert Glück. "A Reversible Programming Language and Its Invertible Self-Interpreter". In: *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. PEPM '07. Nice, France: Association for Computing Machinery, 2007, pp. 144–153. ISBN: 9781595936202. DOI: 10.1145/1244381.1244404. URL: https://doi.org/10.1145/1244381.1244404.