Artifact Documentation for *The Ultimate Conditional Syntax*

For OOPSLA 2024 Artifact Evaluation

Luyu Cheng, Hong Kong University of Science and Technology, China

September 2024

Table of Contents

1. Overview of the Artifact

Our paper introduces a new expressive conditional syntax called *Ultimate Conditional Syntax* (hereinafter referred to as UCS). In the paper, we propose an algorithm to translate this syntax to traditional pattern matching and prove its correctness.

Our artifact implements this syntax and its translation algorithm on the MLscript compiler. The artifacts consists of two parts:

1. *The main project* is a Scala project, which is a complete MLscript compiler, and includes the implementation and tests of UCS;
2. *The web demo* provides a user-friendly interface, allowing people to compile and run MLscript (with UCS) programs directly in the browser, and view the results of each stage of the algorithm described in the paper.

The main project is the paper's main contribution, which fully implements the algorithm specified by the paper. The web demo illustrates the reusability of our main project: it can be reused by other programs (even in different programming languages).

This document contains the evaluation instructions for these two parts (Section 2), explains how the algorithm proposed in the paper was implemented (Section 3), and provides usage instructions for the web demo (Section 4).

1.1. List of Claims

Most claims in the paper can be verified by reproducing particular examples in the test suite. You can test the examples by modifying them directly in the test suite or just trying them out individually in the web, which comes with a syntax reference.

1. **The translation algorithm can handle examples from the paper, and the translated program works correctly after being compiled.** The web demo includes most examples in the paper. You can load them and view the translation intermediate results. See Section 4 for a guide to the web demo.
2. **The translation algorithm's coverage checking stage can identify the three types of redundant branches mentioned in Section 3.5.1 of the paper.** The test files for this claim are located at `shared/src/test/diff/pretyper/ucs/coverage`. The warnings and errors in the output (lines starting with `//|`) after each test block[1] are generated by the translation algorithm.
3. **The translation algorithm's post-processing stage can sort out branches according to Section 3.6 of the paper.** Check the output of following test files:
   - `shared/src/test/diff/pretyper/ucs/stages/PostProcessing.mls`;
   - `shared/src/test/diff/pretyper/ucs/DualOption.mls` since line 172.

[1] Check Section 3.3.1 for more information about test files used in the main project.

1.2. Hardware Requirements

This artifact does not require advanced hardware to run. Any computer connected to the internet and capable of running Java, Node.js, and modern browsers released in recent two years, can complete the artifact evaluation.

1.3. Reusability Guide

The entire main project can be reused in following ways.

1. The main project can be extended directly. One can refer to Section 3 to understand the overall structure of the project. Each stage of the translation in the paper has corresponding source files (as in Section 3.2). People only need to make changes to the relevant parts to extend the translation algorithm as well as other parts of the compiler.
2. The web demo illustrates the other aspect of reusing the main project. In `build.sbt`, we derived a subproject named `npmBuild` from the main project, compiled it to JavaScript via Scala.js, and generated a reusable npm package. This package is referenced in the web demo, thus the main project's compiler can be invoked, and the intermediate results of the UCS translation part are output. Section 2.2.3 exercises the process above.

2. Getting Started

This section gives instructions for setup and basic testing. There are several ways of running the artifacts on your computer. We provide a detailed step-by-step guide for each approach. If you encounter any problems at any step, please contact us on the review platform, and we will provide the most timely assistance.

2.1. Running the Main Project

This section introduces two methods to test running the main project. Please refer Section 3.3 to learn how the tests are organized and performed.

2.1.1. Using Docker

We have built Docker images containing all necessary dependencies compiled for both `amd64` and `arm64` platforms and published them on Docker Hub.

1. Run command `docker run -it --rm mlscript/ucs-docker`[2], which will pull the Docker image, launch a temporary container, run sbt's interactive shell inside the container, and immediately attach to the shell.
2. Run command `mlscriptJVM / test` in the shell, which will run *all* the tests.
3. All tests are expected to be passed.
4. After the test is completed, you can enter `exit` to terminate the sbt shell, and the container just created will be destroyed afterwards.

2.1.2. Starting from Scratch

Before the installation, please make sure that you have the following toolchains installed on your computer.

1. First, you should install **Coursier**, which sets up a Scala development environment. Please follow its instructions according to your operating system. You may need to install Java manually on some platforms.
2. Then, please install **sbt** and **Scala** compiler with Coursier: `cs install sbt scala`.
3. **Node.js 22**. It will also install npm (node package manager), which is also required. If you already have Node.js on your computer but don't want to overwrite the default one, you can consider using nvm (node version manager).

> **Note** Please ensure the `node` executable is in the `PATH`, otherwise all tests will fail.

Next, download and unpack the artifact, and then enter the directory. Suppose the artifact is unpacked at location `artifact/`. Follow the steps below.

1. Run `sbt` at `artifact/`. This will start an interactive shell of sbt.
2. Run command `mlscriptJVM / test` in sbt shell, which will run *all* the tests.

[2] This docker image already includes both `amd64` and `arm64` platforms. Docker will automatically select the image based on your computer's architecture.

3. All tests are expected to be passed.
4. After the test is completed, you can enter `exit` to terminate the sbt shell

2.2. Running the Web Demo

This section only describes how to run the web demo. Please refer to Section 4 for a guidance to the web demo.

2.2.1. Direct Access

The easiest way to start the web demo is to access https://ucs.mlscript.dev in browsers. The web demo is designed to work on desktop computers and laptops. Also, you have to enable JavaScript to use the web demo normally.

Additionally, the web demo is supposed to to work on a modern browser, if you encountered any problems, please first read Section 4.3 for compatible browsers.

> **Note** The website is hosted via Cloudflare Pages statically and does not contain any trackers. Therefore, any access to the web demo will be purely anonymous. Alternatively, you also choose the methods in Section 2.2.2 and Section 2.2.3.

2.2.2. Using Docker

We have built a Docker image specially for running the web demo locally.

1. Run `docker run -d --name web-demo -p 8080:3000 mlscript/ucs-web-demo`.
2. Access http://localhost:8080 with your browser.
3. Remember to delete the container named `web-demo`.

> **Note** You can replace `8080` in the command above with other available ports.

2.2.3. Building from Scratch

If you want to verify that the web demo does use code from the main project, you can build the web demo from scratch by following the steps below. You are supposed to install toolchains mentioned in Section 2.1.2.

1. Assume the artifact has been downloaded and unpacked to the `artifact/`.
2. Run command `sbt "npmBuildJS / Compile / fullLinkJS"` at `artifact/`. This will emit a npm package at `artifact/npm/dist`.
3. Change the directory to `artifact/web-demo/` and run `npm install`. This will install dependencies of the web demo project.
4. Run command `npm run dev`, which will starts a development server. This command will usually have the following output.

```
VITE v5.3.1  ready in 256 ms

➜  Local:   http://localhost:5173/
➜  Network: use --host to expose
➜  press h + enter to show help
```

At this point, you can see the web demo by visiting the suggested URL (usually `http://localhost:5173/`). If you make changes to the source code of the web demo at this time, this development server will automatically update the webpage opened in the browser.

3. Introduction to the Main Project

3.1. Project Structure

The implementation of the main project is based on MLscript. Figure 1 shows the file structure in the project and which source files and tests are related to UCS.

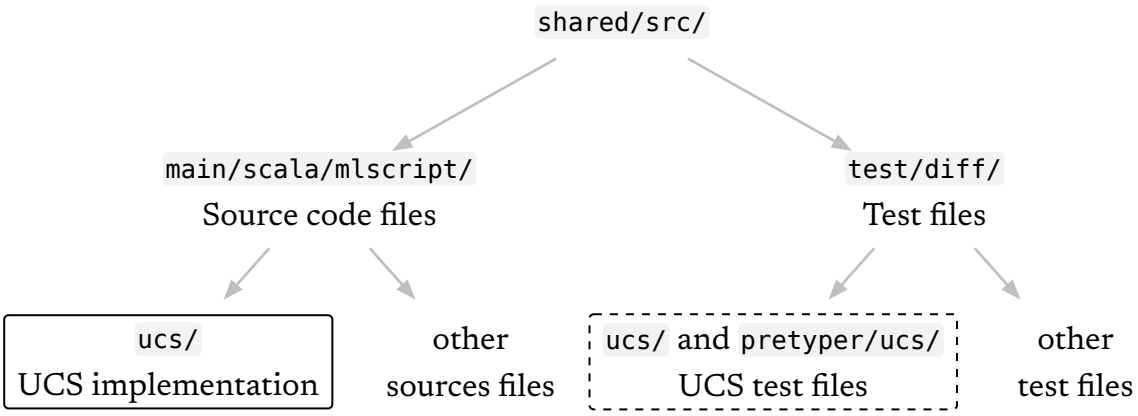

Figure 1: The main project's structure.

The files from framed part is the implementation of the algorithm and the dashed framed part contains the related test files. Section 3.2 describes how these source files correspond to the algorithm in the paper. The purposes of the remaining source files and test files will be described in Section 3.4.

3.2. Correspondence between Paper and Code

The implementation of algorithm and its definition from the paper are implemented in this folder of the artifact: `shared/src/main/scala/mlscript/ucs/`. For brevity, the path names in the following description are relative to this path.

3.2.1. Syntax Definitions

Definition	Location in the Paper	Source Code Files
Source abstract syntax	Fig. 2 in Section 3.1	`syntax/source.scala`
Core abstract syntax	Fig. 3 in Section 3.2	`syntax/core.scala`

3.2.2. Translation Algorithm

Stage	Location in the Paper	Source Code Files
Desugaring	Section 3.3 and Appendix A.1 (Fig. 14 and Fig. 15)	`stages/Desugaring.scala`
Normalization	Section 3.4 and Fig. 8	`stages/Normalization.scala`
Coverage checking	Section 3.5 and Fig. 10	`stages/CoverageChecking.scala`
Post-processing	Section 3.6	`stages/PostProcessing.scala`

Besides the specific stages mentioned above, the class `Desugarer` is responsible for connecting these stages together and integrating them into the entire compiler pipeline.

One may notice that there is a stage `stages/Transformation.scala` that is not introduced in the paper. This is because the parser was implemented before we wrote the paper, therefore the definition of the syntax tree nodes used by the parser is quite different. This stage is designed to convert the old syntax tree nodes to the source abstract syntax nodes described in the paper.

3.3. Running Tests

3.3.1. Test Files Explained

The suffix of the test file is `.mls`. Each file contains several *test blocks* separated by empty lines. For example, the test file in Listing 1 contains three test blocks.

Test output is inserted in place in the test file after each corresponding block, as comments beginning with `//|`. This makes it very easy and convenient to see the test results for each code block. For this reason, we recommend using an editor that automatically reloads open files on changes. VSCode[3] and Sublime Text work well for this.

```
abstract class List[out T]: Cons[T] | Nil                    MLscript
class Cons[out T](head: T, tail: List[T]) extends List[T]
module Nil extends List[nothing]
fun (::) cons(x, xs) = Cons(x, xs)
//| abstract class List[T]: Cons[T] | Nil
//| class Cons[T](head: T, tail: List[T]) extends List
//| module Nil extends List
//| fun (::) cons: forall 'T. ('T, List['T]) -> Cons['T]

fun sum(acc, xs) =
  if xs is
    Cons(x, xs) then sum(acc + x, xs)
    Nil then acc
//| fun sum: (Int, Cons[Int] | Nil) -> Int

sum(0, 1 :: 2 :: 3 :: Nil)
//| Int
//| res
//|     = 6
```

Listing 1: An example of test files.

The output of each test block includes inferred types, evaluation results (by executing transpiled JavaScript programs), and possibly debugging information if debug flags are provided.

[3]We recommend to install mlscript-syntax-highlight extension if using VSCode.

3.3.2. Test in Real-Time with Watch Mode

Using sbt's watch mode, we can modify test files, run tests, and view the test results in real-time. Assume the main project is located at `artifact/`

1. Run sbt at `artifact/`. This starts an interactive shell.
2. Run command `~mlscriptJVM / test`. The leading `~` indicates that the tests will re-run when any test file changes.
3. Open the project with code editor, for example, VSCode. Then, open a UCS test file, for example, `shared/src/test/diff/pretyper/ucs/examples/ListFold.mls`.
4. Make some changes. For example, update line 36 to

   ```
   join(", ")(1 :: 2 :: 3 :: 4 :: Nil)                                MLscript
   ```

   You will see its output at line 40 is updated to

   ```
   //|     = '1, 2, 3, 4'                                            MLscript
   ```

3.3.3. Running Tests Individually

The command given above runs all the tests. Individual tests can be run with option `-z`.

```
~mlscriptJVM/testOnly mlscript.DiffTests -- -z List
```

The command above will watch for file changes and continuously run all `List` tests (those that have "List" in their names). Please note that we have limited the tests to run only those related to UCS, as specified in Section 3.1.

3.4. Other Parts of the Main Project

This section explains the other parts of the main project. The main project has all the basic components of a static-typed programming language compiler: lexer, parser, pre-typer, typer, and code generator.

3.4.1. Lexing and Parsing

The lexer accepts source strings and returns tokens to be parsed. `Lexer.scala` contains the lexer class and `Token.scala` contains the token data types.

The parser accepts tokens generated by the lexer and returns an abstract syntax tree of the input program in the surface syntax. `Parser.scala` contains the parser class and `syntax.scala` contains the *surface* syntax data types of the language.

3.4.2. Pre-Typing

The pre-typing stage is used for name resolution and desugaring of high-level syntaxes (such as UCS) into MLscript's core syntax. The translation algorithm described by the paper is implemented in this stage.

When we packaged this artifact, the pre-typing stage had just been completed, and its name resolution information had not yet been utilized by the typer or the subsequent

code generation stage. As a result, some information can only be displayed with test flag `:ShowPreTyperErrors`.

3.4.3. Type Inference

The `Typer` class accepts an abstract syntax tree of a program and performs type checking. MLscript supports principal type inference with subtyping. Please refer to MLstruct for more information.

`Typer.scala` contains the `Typer` class. `TypeSimplifier.scala` contains type simplification algorithms to simplify inferred types. `TypeDefs.scala` and `NuTypeDefs.scala` contain classes and methods for type declarations. `ConstraitSolver.scala` contains class `ConstraintSolver` which solves subtyping constraints. `NormalForms.scala` the infrastructure to solve tricky subtyping constraints with disjunct normal forms (DNF) on the left and conjunct normal forms (CNF) on the right. `TyperDatatypes.scala` includes data types for internal representation of types with mutable states to support type inference with subtyping. `TyperHelpers.scala` provides helper methods for the typer.

3.4.4. Code Generation

The code generator translates MLscript AST into JavaScript AST and generates the corresponding JavaScript code. Those corresponding files are:

- `codegen/Codegen.scala` contains definitions of JavaScript AST nodes and methods for JavaScript code generation;
- `codegen/Scope.scala` and `codegen/Symbol.scala` provides symbol management and provides hygienic runtime name generation; and
- `JSBackend.scala` contains class `JSBackend` that translates an MLscript AST into a JavaScript AST.

3.5. Compatibility Check

We have tested two methods on the following operating systems. The version of Docker is also annotated. Any similar platform with a proper setup should work as well.

	Using Docker (Section 2.1.1)	**Starting from Scratch** (Section 2.1.2)
macOS 14.5 (Apple Silicon)	Test passed with Docker 26.1.3	Test passed
Windows 11 (x64)	Test passed with Docker 24.0.2	Test passed
Ubuntu 24.04 (x64)	Did Not Test	Test passed
Fedora 40 (x64)	Test passed with Docker 27.0.3	Did Not Test

If you encounter a situation where the project cannot be run, please contact us through the review platform.

4. Guide to the Web Demo

This section provides a hands-on guide for the web demo and illustrates various features of the web demo through screenshots. Most importantly, we have visualized the results of different stages in the paper through the web demo and provided an interactive place to test our translation.

4.1. User Interface

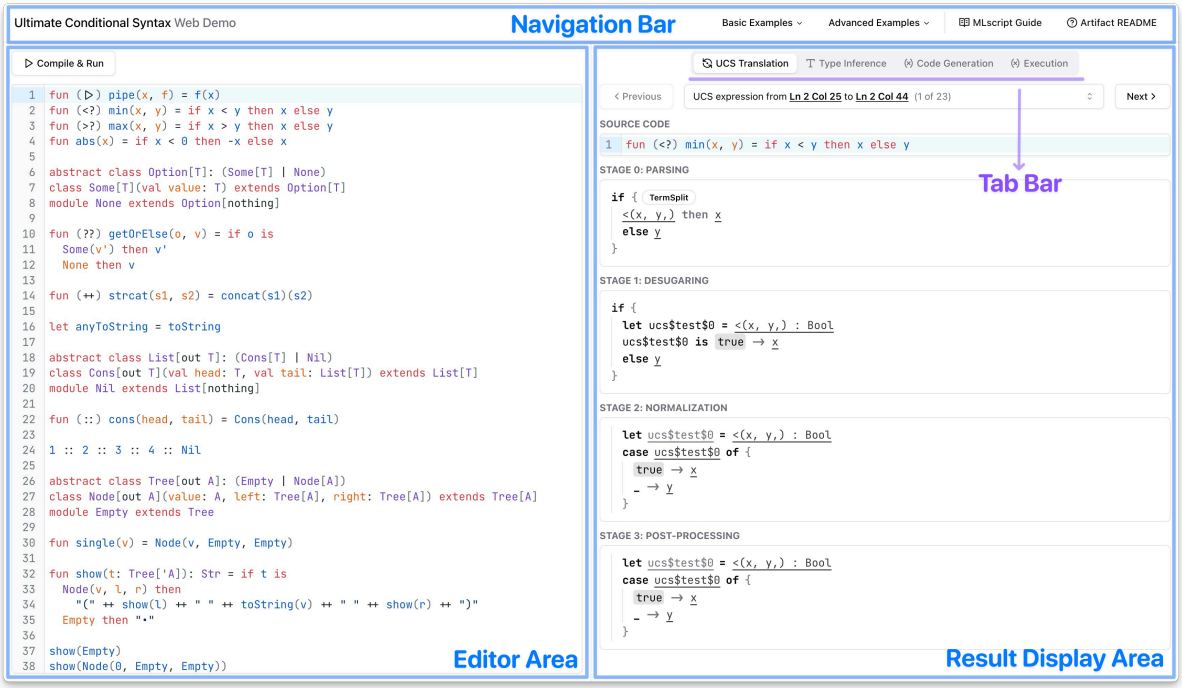

Figure 2: The user interface of the web demo.

As shown in Figure 2, the interface is divided into three parts: *the navigation bar*, *the editor area*, and *the result display area*. Through the navigation bar, we can load existing examples to the editor and view the MLscript tutorials and artifact's README. The code in the editor area is editable, and the "Compile & Run" button at the top will compile the code in the editor into JavaScript and execute it directly in the browser when clicked. The result display area on the right has four tabs, which function as follows.

- The *UCS Translation* tab displays how each UCS expression in the code in the left editor is translated. The paper introduces that the translation of UCS expressions includes three stages: desugaring, normalization, and post-processing. We can see the syntax tree of the expression output at each stage here. Note that these syntax trees are also interactive.
- In the *Type Inference* tab, the results of type inference on the code on the left are displayed, and the panel below shows possible type errors, warnings, and other diagnostic information.
- The *Code Generation* tab displays the JavaScript code transpiled from the MLscript code on the left side.

- The last *Execution* tab shows all the values corresponding to expressions and statements after running the transpiled JavaScript.

We strongly recommend that you check out a few examples and look at the output of each stage of the UCS Translation on the right. We also suggest you read the MLscript guide (on the right side of the navigation bar) and then write some programs using UCS yourself and run them to see the results.

4.2. Built-in Examples

4.2.1. Loading Built-in Examples

The web demo provides many examples from the paper. By hovering the mouse cursor to the navigation buttons “Basic Examples” and “Advanced Examples”, a pop-up menu containing examples will appear (Figure 3).

Click on an example, and it will be loaded into the editor, automatically compiled and executed. Note that the source of each example from the original paper is marked in the lower right corner.

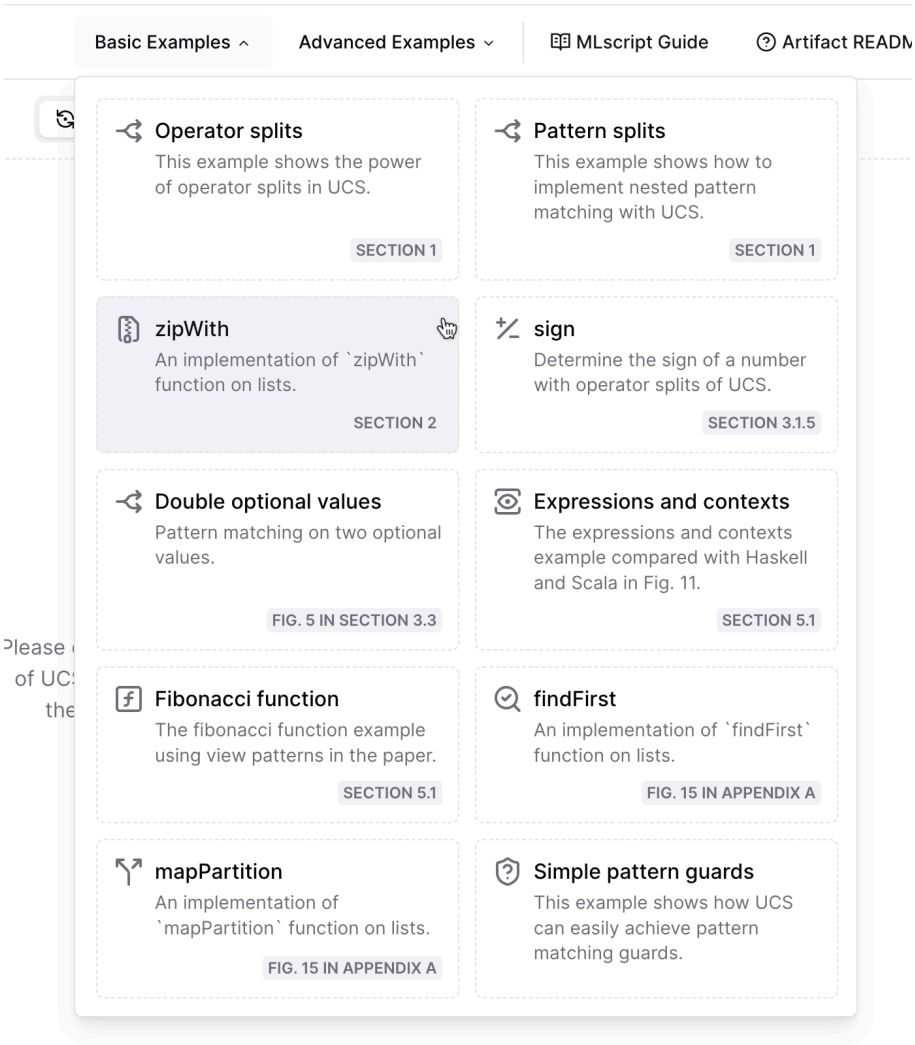

Figure 3: Screenshots of built-in examples in the web demo.

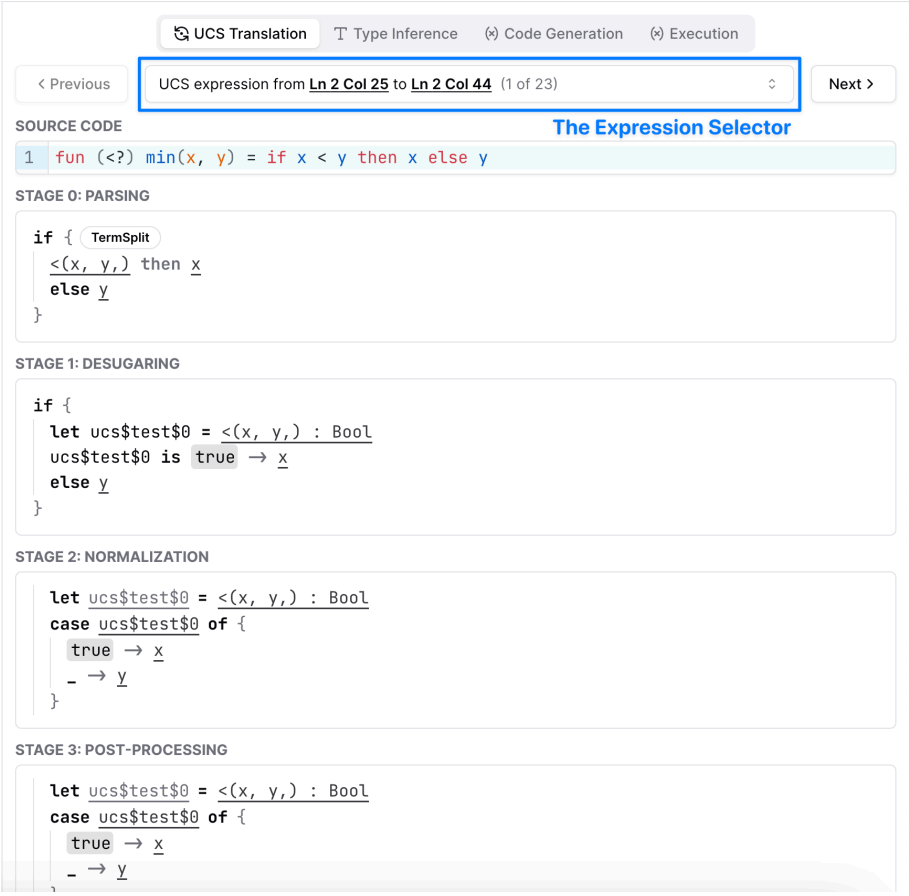

Figure 4: The UCS translation results.

4.2.2. Viewing UCS Translation Results

After an example is loaded, compiled, and executed, you can see the intermediate results of each stage of the UCS translation in the result display area on the right (Figure 4).

The UCS Translation tab displays the intermediate syntax trees from stages of the translation of a UCS expression at a time. Please note that these syntax trees are interactive; for example, users can click on the braces on either side to open/collapse the nested split structures.

From top to bottom, they are:

- the source code in plain text excerpted from the source code;
- the syntax tree obtained after parsing (corresponding to *the source abstract syntax* in the paper), in which we indicate the type of "split" structures;
- the syntax tree obtained after desugaring (corresponding to *the core abstract syntax* in the paper);
- the syntax tree obtained after normalization (corresponding to *the restricted core abstract syntax* in the paper);
- the syntax tree of the final result obtained after post-processing; and
- the coverage checking results.

Since only one UCS expression can be displayed at a time, users can select other UCS expressions from the source code through *the expression selector* (indicated by the blue frame in Figure 4). Users can also click the "Previous" and "Next" buttons on either side to quickly switch expressions.

4.3. Compatibility Check

Generally speaking, people do not need to check their browser version to use the web demo. Since most modern browsers update automatically, any personal computer with a stable internet connection are supposed to run the web demo without issues.

In case you encounter compatibility issues, you can refer to Table 1, which lists the versions of browsers available for the web demo that we have tested on various common operating systems.

	macOS 14	**Windows 11**	**Ubuntu 24.04**	**Fedora 40**
Apple Safari	17.5	N/A	N/A	N/A
Google Chrome	126.0.6478.127	126.0.6478.127	Did Not Test	126.0.6478.126
Microsoft Edge	126.0.2592.87	126.0.2592.87	Did Not Test	126.0.2592.87
Mozilla Firefox	127.0.2	127.0.2	119.0	124.0.1

Table 1: Versions of various browsers that run the web demo across various operating systems which we have tested. They are not the minimum version requirements.

In old browsers (for example, browsers from two years ago), the layout of the web demo may change, and some components may also not display properly. Please forgive us for not being able to support older browsers, and be sure to use the latest browser to access the web demo.