TypeWhich Guide

Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha

July 6, 2021

Contents

Chapter 1

Introduction

TypeWhich is a type migration tool for the gradually-typed lambda calculus with several extensions. Its distinguishing characteristics are the following:

1. TypeWhich formulates type migration as a MaxSMT problem.
2. TypeWhich always produces a migration, as long as the input program is well-scoped.
3. TypeWhich can optimize for different properties: it can produce the most informative types, or types that ensure compatibility with un-migrated code.

Before you read the read of this guide or try to use TypeWhich, we strongly recommend reading Phipps-Costin et al. (2021), which describes TypeWhich depth.

This repository contains the source code for TypeWhich. In addition to the core type migration algorithm, the TypeWhich executable has several auxiliary features:

1. It has a parser for the Grift programming language, which we use to infer types for the Grift benchmarks from Kuhlenschmidt et al. (2019);
2. It has an interpreter for the GTLC, which we use in validation;
3. It has an implementation of the gradual type inference algorithm from Rastogi et al. (2012); and
4. It includes a framework for evaluating type migration algorithms, which we use to compare TypeWhich to several algorithms from the literature Rastogi et al. (2012); Campora et al. (2018); Migeed and Palsberg (2020); Siek and Vachharajani (2008).

Finally, this repository contains several gradual typing benchmarks:

1. The "challenge set" from Phipps-Costin et al. (2021);
2. The benchmarks from Migeed and Palsberg (2020); and
3. The benchmarks from Kuhlenschmidt et al. (2019).

This document will guide you though building TypeWhich, using it on example programs, and using the evaluation framework to reproduce our experimental results.

1.1 Building and Testing TypeWhich

For artifact evaluation, we strongly recommend using the TypeWhich Virtual Machine and skipping this section.

TYPEWHICH is built in Rust and uses Z3 under the hood. In principle, it should work on macOS, Linux or Windows, though we have only tried it on macOS and Linux. *However*, our evaluation uses the implementation from Siek and Vachharajani (2008), which is an old piece of software that is difficult to build on a modern platform. We have managed to compile it a Docker container and produce a 32-bit Linux binary. It should be possible to build it for other platforms, but it will require additional effort. Therefore, **we strongly recommend using Linux to evaluate TYPEWHICH**.

Installing TYPEWHICH Dependencies To build TYPEWHICH from source, you will need:

1. The Rust language toolchain.
2. The Z3 build dependencies and the "usual" build toolchain. On Ubuntu Linux, you can run the following command to get them:

   ```
   sudo apt-get install libz3-dev build-essential
   ```

3. Python 3 and PyYAML to run the integration tests. These are installed by default on most platforms. If you can run the following command successfully then you already have them installed:

   ```
   python3 -c "import yaml"
   ```

Installing Other Type Migration Tools TYPEWHICH does not require these dependencies, but they are necessary to reproduce our evaluation.

1. Migeed and Palsberg (2020) is implemented in Haskell. We have written a parser and printer for their tool that is compatible with TYPEWHICH. This modified implementation is available at the following URL:

   `https://github.com/arjunguha/migeed-palsberg-popl2020`

   Build the tool as described in the repository, and then copy (or symlink) the `MaxMigrate` program to `bin/MaxMigrate` in the TYPEWHICH directory. On Linux, the executable is at:

   ```
   migeed-palsberg-popl2020/.stack-work/install/x86_64-linux-tinfo6/
   lts-13.25/8.6.5/bin/MaxMigrate
   ```

2. Siek and Vachharajani (2008) is implemented in OCaml 3.12 (which is quite old). The following repository has an implementation of the tool, with a modified parser and printer that is compatible with TYPEWHICH:

   `https://github.com/arjunguha/siek-vachharajani-dls2008`

   Build the tool as described in the repository, and then copy (or symlink) the `gtlc` program to `bin/gtubi` in the TYPEWHICH directory.

   Warning: The repository builds a 32-bit Linux executable. You will need to ensure that your Linux system has the libraries needed to run 32-bit code.

3. Campora et al. (2018) The following repository has our implementation of the algorithm from Campora et al. (2018):

   `https://github.com/arjunguha/mgt`

   Build the tool as described in the repository and then copy (or symlink) the the `mgt` program to `bin/mgt` in the TYPEWHICH directory.

   Note: The original implementation by the authors of Campora et al. (2018) does not produce an ordinary migrated program as output. Instead, it produces a BDD that can be interpreted as a family of programs. Our implementation of their algorithm produces programs as output.

Building and Testing Use `cargo` to build TYPEWHICH:

```
cargo build
```

Run the unit tests:

```
cargo test
```

You may see a few ignored tests, but *no tests should fail*.

Test TYPEWHICH using the Grift benchmarks:

```
./test-runner.sh grift grift
```

No tests should fail.

Finally, run the GTLC benchmarks without any third-party tools:

```
cargo run -- benchmark benchmarks.yaml \
  --ignore Gtubi MGT MaxMigrate > test.results.yaml
```

You will see debugging output (on standard error), but the results will be saved to the YAML file. Compare these results to known good results:

```
./bin/yamldiff test.expected.yaml test.results.yaml
```

You should see no output, which indicates that there are no differences.

Chapter 2

Artifact Evaluation: Getting Started

This chapter assumes that you are either:

- Using the TYPEWHICH Virtual Machine, or
- Have installed TYPEWHICH yourself, along with all the third party tools we use for evaluation.

To get started:

1. From the terminal, enter the TYPEWHICH directory:

   ```
   cd typewhich
   ```

2. Run the TYPEWHICH benchmarks, and output results to `results.yaml`:

   ```
   ./bin/TypeWhich benchmark > results.yaml
   ```

   This will take less than five minutes to complete. This command runs the benchmark programs using five tools (and TYPEWHICH in two modes). For each benchmark, you will thus see six lines of output (on standard error):

   ```
   Running Gtubi on adversarial/01-farg-mismatch.gtlc ...
   Running InsAndOuts on adversarial/01-farg-mismatch.gtlc ...
   Running MGT on adversarial/01-farg-mismatch.gtlc ...
   Running MaxMigrate on adversarial/01-farg-mismatch.gtlc ...
   Running TypeWhich2 on adversarial/01-farg-mismatch.gtlc ...
   Running TypeWhich on adversarial/01-farg-mismatch.gtlc ...
   ```

   The `InsAndOuts` tool does not terminate on three benchmarks, and we kill it after some time. So, you will see `Killed` three times in the output. This is expected.

3. Check that the results are identical to known good results:

   ```
   ./bin/yamldiff expected.yaml results.yaml
   ```

   You should see no output, which indicates that there are no differences.

FILL Grift benchmarks.

At this point, it should be possible to validate the results in depth.

Chapter 3

Artifact Evaluation: Step by Step Guide

This chapter assumes you have completed the steps in Chapter 2.

3.1 Claims To Validate

The paper makes the following claims that can be validated:

1. Figure 15 reports the results of several type migration tools on a a suite of benchmarks. Specifically, it categorizes them into several columns. This artifact generates the figure, and the raw data and data analysis scripts can be validated.

2. Section 6.5 runs TypeWhich on benchmarks written in Grift. These benchmarks have two versions: one that has no type annotations, and the other that has human-written type annotations. When run on the unannotated Grift benchmarks, TypeWhich calculates all but two of the human-written annotations.

3. Section 6.6 reports that our full suite of benchmarks is 892 LOC, and TypeWhich takes three seconds to run on all of them. It will take longer in a virtual machine, but should be roughly the same. i.e., it will be significantly less than 30 seconds.

The rest of this section will walk you through validating these results.

3.1.1 GTLC Benchmarks on Multiple Tools

In the previous chapter, we generated `results.yaml`. That ran TypeWhich and all other tools on two suites of benchmarks:

1. The `migeed` directory contains the benchmarks from Migeed and Palsberg (2020) written in the concrete syntax of TypeWhich.

2. The `adversarial` directory contains the “challenge set” from the TypeWhich paper.

The evaluation framework is driven by the file `benchmarks.yaml`, which specifies a list of type migration tools at the top, and is followed by a list of benchmark files, with some additional information. The entire benchmarking procedure is implemented in `src/benchmark.rs`:

1. It checks that the tool produces valid program, to verify that the tool did not reject the program.

2. It runs the original program and the output of the tool and checks that they produce the same result, to verify that the tool did not introduce a runtime error.

3. In a gradually typed language, increasing type precision can make a program incompatible with certain contexts. To check if this is the case, every benchmark in the YAML file *may* be accompanied by a context that witnesses the incompatibility: the framework runs the original and migrated program in the context, to check if they produce different results.

4. The framework counts the number of `any`s that are eliminated by the migration tool. Every eliminated `any` improves precision, but *may or may not* introduce an incompatibility, but this requires human judgement. For example, in the program `fun x . x + 1`, annotating “x” with `int` does not introduce an incompatibility. However, in `fun x . x`, annotating “x” with `int` is an incompatibility. The framework flags these results for manual verification. However, it allows the input YAML to specify expected outputs to suppress these warnings when desired.

The file `results.yaml` is a copy of `benchmarks.yaml` with output data added by the benchmarking framework. We use this file to generate Figure 15 in the paper. You should validate that table as follows:

1. Check that `results.yaml` does not have any errors: look for the string “Disaster” in that file. It should not occur!

2. Regenerate the LaTeX snippet for the table with the following command:

   ```
   ./bin/TypeWhich latex-benchmark-summary results.yaml
   ```

   The output that you will see will see is roughly the LaTeX code for Figure 15, with two small differences: it prints `TypeWhich2` instead of `TypeWhichC` and `TypeWhich` instead of `TypeWhichP`. However, the order of rows and columns is exactly the same as the table in the paper. It should be straightforward to check that the fractions in this output are exactly the fractions reported in the table.

3.1.2 Grift Benchmarks with TypeWhich

[FILL]

3.1.3 Performance

From the TypeWhich directory, run the following command:

```
time ./performance.sh
```

The script will take roughly three seconds to complete. You can read the script to verify that it runs TypeWhich on three suites of benchmarks:

1. `migeed/*.gtlc`: the benchmarks from Migeed and Palsberg (2020),

2. `adversarial/*.gtlc`: the “challenge set” from our paper, and

3. `grift-suite/benchmarks/src/dyn/*.grift`: the benchmarks from Kuhlenschmidt et al. (2019).

3.2 Exploring Type Migrations

Our artifact includes several type migration tools, in addition to TypeWhich, and we have hacker their parsers to work with the same concrete syntax, so that it is easy to use any tool on the same program. We encourage you to try some out, and to modify the benchmarks as well. Here are the available tools:

- To run Migeed and Palsberg (2020):

  ```
  ./bin/MaxMigrate FILENAME.gtlc
  ```

- To run Campora et al. (2018):

```
./bin/mgt FILENAME.gtlc
```

- To run Siek and Vachharajani (2008):

```
./bin/gtubi FILENAME.gtlc
```

- To run Rastogi et al. (2012):

```
./bin/TypeWhich migrate --ins-and-outs FILENAME.gtlc
```

- To run TYPEWHICH and produce types that are safe in all contexts:

```
./bin/TypeWhich migrate FILENAME.gtlc
```

- To run TYPEWHICH and produce precise types that may not work in all contexts:

```
./bin/TypeWhich migrate --precise FILENAME.gtlc
```

Example Create a file called `input.gtlc` with the following contents:

```
(fun f. (fun y. f) (f 5)) (fun x. 10 + x)
```

This program omits all type annotations: TYPEWHICH assumes that omitted annotations are all **`any`**. We can migrate the the program using TYPEWHICH in two modes:

1. In *compatibility mode*, TYPEWHICH infers types but maintains compatibility with un-migrated code:

```
$ ./bin/TypeWhich migrate input.gtlc
(fun f:any -> int. (fun y:int. f) (f 5)) (fun x:any. 10 + x)
```

2. In *precise mode*, TYPEWHICH infers the most precise type that it can, though that may come at the expense of compatibility:

```
$ ./bin/TypeWhich migrate --precise inpuy.gtlc
(fun f:int -> int. (fun y:int. f) (f 5)) (fun x:int. 10 + x)
```

3.3 Input Language

TYPEWHICH supports a superset of the GTLC, written in the following syntax. Note that the other tools do not support all the extensions documented below.

b	:=	**true** | **false**	Boolean literal
n	:=	... | -1 | 0 | 1 | ...	Integer literals
s	:=	"..."	String literals
c	:=	b | n | s	Literals
T	:=	**any**	The unknown type
	|	**int**	Integer type
	|	**bool**	Boolean type
	|	T_1 **->** T_2	Function type
	|	**(** T **)**	
e	:=	x	Bound identifier
	|	c	Literal
	|	e **:** T	Type ascription
	|	**(** e **)**	Parenthesis
	|	**fun** x **.** e	Function
	|	e_1 e_2	Application
	|	e_1 **+** e_2	Addition
	|	e_1 ***** e_2	Multiplication
	|	e_1 **=** e_2	Integer equality
	|	e_1 **+?** e_2	Addition or string concatenation (overloaded)
	|	**(**e_1**,**e_2**)**	Pair
	|	**fix** f **.**e	Fixpoint
	|	**if** e_1 **then** e_2 **else** e_3	Conditional
	|	**let** x **=** e_1 **in** e_2	Let binding
	|	**let rec** x **=** e_1 **in** e_2	Recursive let binding

Bibliography

J. P. Campora, S. Chen, M. Erwig, and E. Walkingshaw. Migrating gradual types. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL), Dec. 2018.

A. Kuhlenschmidt, D. Almahallawi, and J. G. Siek. Toward efficient gradual typing for structural types via coercions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019.

Z. Migeed and J. Palsberg. What is decidable about gradual types? *Proceedings of the ACM on Programming Languages (PACMPL)*, 4(POPL), Dec. 2020.

L. Phipps-Costin, C. J. Anderson, M. Greenberg, and A. Guha. Solver-based gradual type migration. `https://khoury.northeastern.edu/~arjunguha/main/papers/2021-typewhich.html`, 2021. In submission.

A. Rastogi, A. Chaudhuri, and B. Hosmer. The ins and outs of gradual type inference. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.

J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Dynamic Languages Symposium (DLS)*, 2008.