TypeWhich Guide

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

May 4, 2021

Contents

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.

For more information on TypeWhich, see Phipps-Costin et al. (2021).

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.

2 Building and Testing TYPEWHICH

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) [FILL]

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

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.

3 Running TYPEWHICH

The TYPEWHICH executable is symlinked to `bin/TypeWhich`. TYPEWHICH expects its input program to be in a single file, and written in either Grift (extension `.grift`) or in a superset of the gradually typed lambda calculus (extension `.gtlc`), shown in Section 4.

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)
```

The TYPEWHICH executable supports several other sub-commands and flags. Run `./bin/TypeWhich -help` for more complete documentation.

4 Input Language

`./doc/doc.pdf` has the same content as this file, but with slightly better formatting.

TYPEWHICH supports a superset of the GTLC, written in the following syntax:

[FILL] A few cases missing

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

5 Evaluation Framework

To run the full suite of experiments, you will need to install the third-party type migration tools.

TYPEWHICH includes a framework for evaluating type migration algorithms, which is driven by a YAML that specifies a list of type migration tools to evaluate, and benchmark programs for the evaluation. The framework runs every tool on every benchmark and then validates the result as follows:

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, *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 `./benchmarks.yaml` drives the evaluation framework to compare TYPEWHICH and several other type migration algorithms on a suite of benchmarks. We recommend reading that file to understand benchmarking in more depth.

To run the experiments, use the following command:

```
./bin/TypeWhich benchmark benchmarks.yaml > RESULTS.yaml
```

It prints progress on standard error. The output is a YAML file of results.

5.1 Validation

1. The benchmarking script does a lot of validation itself.
2. In `RESULTS.yaml`, look for the string "Disaster". It should not appear!
3. In `RESULTS.yaml`, look for the string `manually_verify`. These are results from experiments where (1) we could not crash the migrated program, and (2) the migrated program has fewer 'any's than the original. So, the table of results counts this migration as one that is 100% compatible with untyped contexts. But, it requires a manual check.
4. Finally, you can compare `RESULTS.yaml` with a known good output from benchmarking:

   ```
   ./bin/yamldiff RESULTS.yaml expected.yaml
   ```

5.2 Results

To generate the summary table found in Phipps-Costin et al. (2021), use the following command:

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

To generate the appendix of results:

```
./bin/TypeWhich latex-benchmarks RESULTS.yaml
```

6 Benchmarks

The TYPEWHICH repository has several benchmarks:

1. The `migeed` directory contains the benchmarks from Migeed et al., written in the concrete syntax of TYPEWHICH.
2. The `adversarial` directory contains the "challenge set" from the TYPEWHICH paper.
3. The `grift-suite` directory contains tests from Grift. The `mu/` directory has been modified to use Dyn where it originally used recursive types.
4. The `grift-suite/benchmarks` contains benchmarks from `https://github.com/Gradual-Typing/benchmarks` with the following adjustments:
   (a) The getters and setters in n-body have been removed. They were neither used nor exported we opted to remove these functions from the benchmark. This is discussed in the paper.
   (b) We have changed where in the program some benchmarks print a terminating newline for consistency between the static and dynamic versions.
   (c) Benchmarks that rely on modules are removed

References

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.