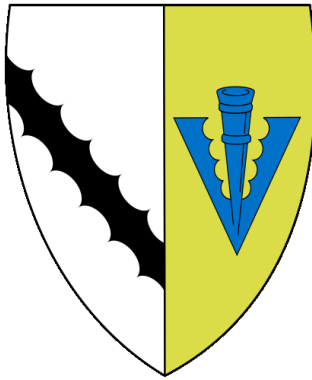


Author Here

Title Here

Tripes & Part Here



College Here College
University of Cambridge
October 22, 2024

*A dissertation submitted to the University of Cambridge in
partial fulfilment for a Bachelor of Arts*

Declaration of Originality

Declaration Here.

Proforma

Candidate Number: **Candidate Number Here**
College: **College Here**
Project Title: **Title Here**
Examination: **Tripod & Part Here—**
MM/YYYY Submission Month
Deadline Here
Word Count: **34** ¹
Code Line Count: **Code Count** ²
Project Originator: **Project Originator Here**
Supervisors: **Supervisor 1, Supervisor 2**

Original Aims of the Project

Aims Here. Concise summary of proposal description.

Work Completed

Work completed by deadline.

¹ *Calculated by `texcount`. Including: tables and footnotes. Excluding: the front matter, bibliography, and appendices*

² *Calculation method here*

Special Difficulties

Any Special Difficulties encountered

Acknowledgements

Acknowledgements Here.

Contents

1	Introduction	1
2	Preparation	2
3	Implementation	3
4	Evaluation	4
5	Conclusions	5
	Bibliography	6
A	Formal Semantics and Proofs	8
B	Another Appendix	9
	Index	10
	Project Proposal	11
	Description	11
	Starting Point	13
	Success Criteria	13
	Work Plan	15
	Resource Declaration	18

Chapter 1

Introduction

Introduction here.

Example citation [\[7\]](#). Example index test

Chapter 2

Preparation

Preparation Here.

Chapter 3

Implementation

Implementation Here.

Chapter 4

Evaluation

Evaluation Here.

Chapter 5

Conclusions

Conclusions Here.

Bibliography

- [1] *Hazel Project Website*. URL: <https://hazel.org/>.
- [2] *Hazel Source Code*. URL: <https://github.com/hazeltgrove/hazel>.
- [3] Bogdan Korel and Janusz Laski. “Dynamic program slicing”. In: *Information Processing Letters* 29.3 (Oct. 1988), pp. 155–163. ISSN: 0020-0190. DOI: [10.1016/0020-0190\(88\)90054-3](https://doi.org/10.1016/0020-0190(88)90054-3). URL: [http://dx.doi.org/10.1016/0020-0190\(88\)90054-3](http://dx.doi.org/10.1016/0020-0190(88)90054-3).
- [4] Cyrus Omar et al. “Live functional programming with typed holes”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (Jan. 2019), pp. 1–32. ISSN: 2475-1421. DOI: [10.1145/3290327](https://doi.org/10.1145/3290327). URL: <http://dx.doi.org/10.1145/3290327>.
- [5] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. “Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong)”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP’16. ACM, Sept. 2016, pp. 228–242. DOI: [10.1145/2951913.2951915](https://doi.org/10.1145/2951913.2951915). URL: <http://dx.doi.org/10.1145/2951913.2951915>.
- [6] F. Tip and T. B. Dinesh. “A slicing-based approach for locating type errors”. In: *ACM Transactions on Software Engineering and Methodology* 10.1 (Jan. 2001), pp. 5–55.

ISSN: 1557-7392. DOI: [10.1145/366378.366379](https://doi.org/10.1145/366378.366379). URL: <http://dx.doi.org/10.1145/366378.366379>.

- [7] Philip Wadler and Robert Bruce Findler. “Well-Typed Programs Can’t Be Blamed”. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2009, pp. 1–16. ISBN: 9783642005909. DOI: [10.1007/978-3-642-00590-9_1](https://doi.org/10.1007/978-3-642-00590-9_1). URL: http://dx.doi.org/10.1007/978-3-642-00590-9_1.

Appendix A

Formal Semantics and Proofs

Example Appendix Here

Appendix B

Another Appendix

Another Example Appendix Here

Index

test index description here,

[1](#)

Project Proposal

Description

This project will add some features to the Hazel language [1]. Hazel is a functional research language that makes use of gradual types to support unusual features such as: holes (code placeholders) to give type meaning to incomplete programs. Importantly for this project, all Hazel programs, even ill-typed or incomplete programs, are evaluable. This allows dynamic reasoning about ill-typed programs via evaluation traces with the potential to improve the user's understanding of *why ill-typed programs go wrong*. See example below:

```
let rec sum : Int -> Int = fun n ->
  if n == 0 then
    true      // Type error statically caught and correctly located
  else
    n + sum(n - 1)
in sum(2)
```

But evaluation is still possible; see below a (compressed) trace to a stuck value exhibiting a cast error:

$$\text{sum}(2) \mapsto^* 2 + \text{sum}(1) \mapsto^* 2 + (1 + \text{true}^{\langle \text{Bool} \Rightarrow \text{Int} \rangle})$$

This project aims to exploit further this potential by providing some extra features to both: aid with finding values/inputs

that demonstrate why type-errors were found (type-error witnesses) and linking the evaluation traces back to source code. But is not expected to directly measure the usefulness of such evaluation traces themselves in debugging, nor is the design space for a Hazel debugger inspecting and interacting with traces to be explored.

Searching for type-error witnesses automatically is the main feature provided by this project, inspired by Seidel et al. [5]. The intended use of this is to automatically generate values (for example, function arguments) that cause ill-typed programs to ‘go wrong’ (lead to a cast error). More specifically, the search procedure can be thought of as evaluating a *special hole* which refines its type dynamically and non-deterministically instantiates itself to values of this type to find a value whose evaluation leads to a *general* cast error – ‘general’ meaning excluding trivial cast errors such as generating a value that doesn’t actually have the refined expected type.

Such a search procedure is undecidable and subject to path explosion, hence the success criteria (detailed below) does not expect witnesses to be provided in general, even if they do exist. Sophisticated heuristics and methods to limit path explosion to support large code samples is not a core goal.

Formal semantics of this procedure and associated proofs is an extension goal, consisting of preservation proofs and witness generality proofs (formalising the notion of generality mentioned previously).

Secondly, *cast slicing* will track source code that contributed to any cast throughout the cast elaboration and evaluation phases. In particular, this allows a cast involved in a cast error relating to a type-error witness to point back to offending code. This is expected in some sense to be similar to blame tracking [7], error and dynamic program slicing [6, 3], although these are not directly relevant for this project.

Work required for the creation of an evaluation corpus of

ill-typed hazel programs, requiring manual work or creation of automated translation and/or fuzzing tools, is timetabled.

Starting Point

Only background research and exploration has been conducted. This consists of reading the Hazel research papers [1] and various other related research topics including: gradual types, bidirectional types, symbolic evaluation, OCaml error localisation and visualisation techniques.

More research, into the Hazel codebase in particular, and concrete planning is required and is timetabled accordingly.

Success Criteria

Core goals are the minimum expected goals that must be completed to consider this project a success. This corresponds to a working tool for a large portion of Hazel.

Extension goals will be timetabled in, but are relatively more difficult and not required for the project to be considered a success.

First, I give some definitions of terms:

- **Core Calculus** – The formal semantics core of Hazel as referred to by the Hazel research papers [4].
- **Basic Hazel** – A Hazel subset consisting of the core calculus, product and sum types, type aliases, bindings, (parametric) lists, bools, int, floats, strings, and their corresponding standard operations.
- **Full Hazel** – Hazel, including **Basic Hazel** plus pattern matching, explicit impredicative system-F style polymorphism and explicitly recursive types.

- **Core Corpus** – A corpus of ill-typed Hazel programs that are similar in complexity and size to student programs being taught a functional language, *e.g.* (*incorrect*) *solutions to the ticks in FoCS*. This will include examples in **Basic** or **Full Hazel** as required.
- **Extended Corpus** – A corpus of ill-typed Hazel programs that are larger in size, more akin to real-world code.
- **Evaluation Criteria** – Conditions for the search procedure to meet upon evaluation:
 1. Must have reasonable coverage – success in finding an *existing* witness which is correct and general.
 2. Must find witnesses in an amount of time suitable for interactive debugging – in-line with build-times for a debug build of existing languages.

Core Goals

- Success criteria for Cast Slicing – Cast slicing must be *correct* (slices must include all code involved in the cast) and work for *all casts*, including casts involved in cast errors. Informal reasoning in evidence of satisfying these conditions is all that will be required.
- Success criteria for the Search Procedure – The procedure must work for **Basic Hazel**, meeting the **Evaluation Criteria** over the **Core Corpus**. Analysis of some classes of programs for which witnesses could not be generated is also expected.

Extension Goals

- Search Procedure Extensions – Support for **Full Hazel** under the same criteria as above.
- Search Procedure Performance Extensions – Meeting of the **Evaluation Criteria** over an **Extended Corpus**
- Formal Semantics – The specification of a formal evaluation semantics for the search procedure over the **Core Calculus**. Additionally, a preservation and witness generality proof should be provided.

Work Plan

21st Oct (*Proposal Deadline*) – 3rd Nov

Background research & research into the Hazel semantics, cast elaboration, type system, and codebase. Produce implementation plan for cast slicing and the search procedure for the **Core Calculus**. This includes an interaction design plan, expected to be very minimal.

Milestone 1: Plan Confirmed with Supervisors

4th Nov – 17th Nov

Complete implementation of Cast Slicing for the **Core Calculus**. Write detailed reasoning for correctness, including plan for **Basic Hazel**. Add unit testing.

*Milestone 2: Cast slicing is complete for the **Core Calculus**.*

18th Nov – 1st Dec (*End of Full Michaelmas Term*)

Complete implementation of the search procedure for the **Core Calculus**.

*Milestone 3: Search Procedure is complete for the **Core Calculus**.*

2nd Dec – 20th Dec

Extension of both cast slicing and the search procedure to **Basic Hazel**.

*Milestone 4: Cast slicing & search procedure are complete for **Basic Hazel***

21st Dec – 24th Jan (*Full Lent Term starting 16th Jan*)

Basic UI interaction for the project. Drafts of Implementation chapter. Slack time. Expecting holiday, exam revision, and module exam revision. Should time be available, the **Formal Semantics** extension will be attempted.

Milestone 5: Implementation chapter draft complete.

25th Jan – 7th Feb (*Progress Report Deadline*)

Writing of Progress Report. Planning of evaluation, primarily including decisions and design of tools to be used to collect/create the **Core Corpus** and planning the specific statistical tests to conduct on the corpus. Collected corpus and translation method will be one of:

1. Manual translation of a small ill-typed OCaml program corpus into ill-typed Hazel.
2. Manual insertion of type-errors into a well-typed Hazel corpus.
3. Collection of a well-typed Hazel corpus.
Tools: A Hazel type fuzzer to make the corpus ill-typed.
4. Collection of a well-typed OCaml corpus.
Tools: OCaml -i Hazel translator/annotator which works with well-typed OCaml. A Hazel type fuzzer.
5. Collection of an ill-typed OCaml corpus.
Tools: OCaml -i Hazel translator which works with ill-typed OCaml. *This would NOT be expected to be an implicitly typed Hazel front-end which maintains desirable properties like parametricity.*

Milestone 6: Evaluation plan and corpus creation method confirmed with supervisors.

Milestone 7: Underlying corpus (critical resource) collected.

8th Feb – 28th Feb

Implementation of the required tools for evaluation as planned. Some existing code or tools may be re-used, such as the OCaml type-checker.

*Milestone 8: **Core Corpus** has been collected.*

1st Mar – 15th Mar (*End of Full Lent Term*)

Conducting of evaluation tests and write-up of evaluation draft including results.

Milestone 9: Evaluation results documented.
Milestone 10: Evaluation draft complete.

16th Mar – 30th Mar

Drafts of remaining dissertation chapters. If possible, collection and evaluation of **Extended Corpus** using the same tools as the **Core Corpus**.

Milestone 11: Full dissertation draft complete and sent to supervisors for feedback.

31st Mar – 13th Apr

Act upon dissertation feedback. Exam revision.

Milestone 12: Second dissertation draft complete and send to supervisors for feedback.

14th Apr – 23rd Apr (*Start of Full Easter Term*)

Act upon feedback. Final dissertation complete. Exam revision.

Milestone 13: Dissertation submitted.

24th Apr – 16th May (*Final Deadline*)

Exam revision.

Milestone 14: Source code submitted.

Resource Declaration

- Underlying Corpus of either: Well-typed OCaml programs, Ill-typed OCaml programs, Hazel programs. For

use in evaluation. The required tools or manual translation to convert these into the ill-typed Hazel **Core Corpus** are detailed and allocated time in the timetable.

- Hazel source code. Openly available with MIT licence on GitHub [2].
- My personal laptop will be used for development, using GitHub for version control and backup of both code and dissertation. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. A backup pc is available in case of such failure.