THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

Implementation of fill-and-resume in the Hazel live programming environment

by

Jonathan Lam

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Engineering

Professor Fred L. Fontaine, Advisor

Professor Robert Marano, Co-advisor

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART

ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

Barry L. Shoop, Ph.D., P.E. Date

Fred L. Fontaine, Ph.D. Date

ACKNOWLEDGEMENTS

TODO

ABSTRACT

TODO

TABLE OF CONTENTS

LIST OF FIGURES

TABLE OF NOMENCLATURE

TODO

1 Introduction

1.1 Functional programming

1.1.1 Recursion and mathematical induction

1.1.2 The λ-calculus

1.1.3 Purity and statefulness

1.1.4 Comparison to other programming paradigms

1.2 Classifications of type systems

1.2.1 Primer on programming language semantics and reasoning

1.2.2 Static vs. dynamic typing

1.2.3 Strong vs. weak typing

1.2.4 Inferred vs. manifold typing

1.2.5 Gradual typing

1.2.6 Other classifications

1.3 Approaches to programming interfaces

1.3.1 Structure editors

1.3.2 Graphical editors

1.3.3 Intentional, generative, and meta-programming

1.3.4 Applications to programming education

1.3.5 Drawbacks of non-textual editors

2 An overview of the Hazel programming environment

2.1 Hazelnut static semantics

2.1.1 Expression and type holes

2.1.2 Bidirectional typing

2.1.3 Example of bidirectional type derivation

2.2 Hazelnut Live dynamic semantics

2.2.1 Example of elaboration

2.2.2 Example of evaluation

2.2.3 Example of hole instance numbering

2.3 Hazel programming environment

2.3.1 Explanation of interface

2.3.2 Implications of Hazel

3 Problem statement and related works

3.1 Problem statement

3.2 Issues with the current implementation

3.2.1 Hole numbering inefficiencies

3.2.2 Hole instance tracking inefficiencies

3.3 CMTT interpretation of fill-and-resume

3.4 Notebook-style live programming environments

4 Implementation and optimization of FAR

4.1 Evaluation with environments

4.1.1 Evaluation with substitution vs. with environments

Evaluation in Hazel was performed using originally using a *substitution model of evaluation*, which is a theoretically simpler model. In this model, variables that are bound by some construct are substituted into the construct's body. For example, the variable(s) bound using a `let`-expression pattern are substituted in the `let`-expression's body, and the variable(s) bound during a function application are substituted into the function's body, and then the body is evaluated.

In this formulation, variables are "given meaning" via substitution; once evaluation reaches an expression, all variables in scope (in the typing context) will have been replaced by their value by some containing binding expression. In other words, variables are never evaluated directly; they are substituted by their values when bound, and their values are evaluated. The substitution model is useful for teaching purposes because it is simple and close to its mathematical definition: a variable can be thought of as an equivalent stand-in for its value.

However, for the purpose of computational efficiency, a model in which values are lazily expanded ("looked-up") only when needed is more efficient. This is called the *environment model of evaluation*, and generally is more efficient because the runtime does not need to perform an extra substitution pass over subexpressions; untraversed (unevaluated) branches do not require substituting; and the runtime does not need to carry an expression-level IR of the language. The last point is due to the fact that the substitution model manipulates expressions, while evaluation does not; this means that the latter is more amenable for compilation, and is how compiled languages tend to be implemented: each frame of the theoretical stack frame is a de facto environment frame. While switching from the substitution to environment model is not an improvement in asymptotic efficency, these effects are useful especially for high-performance and compiled languages.

In the case of Hazel (which does not prioritize speed of evaluation in its implementation, and is not a compiled language), evaluation with (reified) environments offers an additional (performance) benefit over the substitution model: the ability to easily identify (and thus memoize) operations over environments. This is useful for the optimizations described later in this paper.

Note that the substitution model does not imply a lazy (i.e., normal-order, call-by-name, call-by-need) evaluation as in languages such as Haskell or Miranda, in which bound variables are (by default) not evaluated until their value is required. Laziness is conceptually tied to substitution, but the substitution model does not require laziness. Like most programming languages, Hazel only has strict (i.e., applicative-order, call-by-value) evaluation: the expressions bound to variables are evaluated at the time of binding.

The implementation of evaluation with environments differs from that of evaluation with substitution primarily in that: an evaluation environment is required to look up bound variables as evaluation reaches them; binding constructs extend the evaluation environment rather than performing substitution; and λ abstractions are bound with their evaluation environment at runtime to form (lexical) closures.

The implementation of evaluation in Hazel differs from a typical interpreter implementation of evaluation with environments in three regards: we need to account for hole environments; environments are uniquely identified by an identifier for memoization (in turn for optimization); and any closures in the evaluation result should be converted back into plain λ abstractions, for reasons that will be discussed later TODOREF.

4.1.2 Formalization of evaluation with environments

Omar et al. [1] describes evaluation with the substitution model using a little-step semantics with an evaluation context $\mathcal{E}$, reproduced in TODOREF.

TODOREF is an analogous small-step description of the substitution model, also using the little-step semantics.

The Hazel implementation follows a big-step evaluation model, so a big-step formalization is also displayed in TODOREF.

4.2 Restructuring hole numbering

4.3 Implementing FAR

4.4 Memoization of recent actions

4.5 UI changes for notebook-like editing

5 Evaluation of FAR

6 Conclusions and recommendations

REFERENCES

[1] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *PACMPL*, 3(POPL), 2019.

APPENDICES

A Formalization of evaluation with environments

B Contributions to Hazel

B.1 Equivalent evaluation with environments

B.2 Identifying performance issue

B.3 Simplifying hole renumbering process

B.4 Implementation of FAR functionality

B.5 Memoization for FAR

B.6 Additional performance improvements

C Selected code samples