Zeus

Algorithmic Program Equivalence

Vijay Ramamurthy
Advisor: Umut A. Acar

Senior Honors Thesis

School of Computer Science
Carnegie Mellon University
2019

Abstract

Course staffs often grade student code by hand to gauge fine-grained properties that are hard to autograde such as runtime, what approach the student took, and how conceptually similar the student's approach was to a correct one. The primary drawback of hand-grading is the amount of the course staff's time it takes, especially when code is conceptually complicated and when the course size is large; in turn, the size of the course staff limits the conceptual complexity of coding assignments and the number of students who can be allowed to enroll in the course. In practice, for any paricular approach a student takes when writing a piece of code, there tend to be many other students who take the same approach. Zeus checks programs for equivalence and uses that to automatically partition student code into "equivalence class" buckets, where every student who took a particular approach is placed in a bucket with all other students who took the same approach. A course staff would then only need to hand-grade one submission per bucket, which saves time if the equivalence checker is good enough for the number of buckets to be lower than the number of students.

Contents

Chapter 1

Introduction

As enrollment in computer science courses rises, the question of how to grade programming assignments becomes more pressing. When grading programming assignments, two forces have always been at odds: the desire to provide detailed feedback specific to each student's code, and the desire to grade quickly.

At one end of the spectrum is hand-grading. When a (human) grader takes the time to read through and understand a student's code, they can reason about how similar the approach taken by the student is to that of the reference implementation. Further, if there are many reference implementations then after reasoning a bit the grader can figure out which reference implementation the student's code is most similar to, and whether it exactly matches any of them. When grading many pieces of code a particularly adept grader may even identify common mistakes by recognizing when a student's incorrect implementation takes the same approach as a previously seen incorrect implementation. With infinite mental resources a grader could in theory remember which students took which common correct/incorrect approach (if any) and give feedback specific to that approach. Since reasoning about code by hand allows a grader to draw such fine-grained conclusions about the structure of the code, hand-grading can be used to grade students to arbitrarily fine-grained criteria such as asymptotic complexity, stylistic structuring of code, etc.

Of course, while hand-grading provides really good detailed feedback specific to how students structured their code, it's impractical with limited course staff sizes and ever-increasing enrollment. To hand-grade programming assignments would either take exorbitant amounts of grader time or would require the assignments to be severely limited in sophistication and length. A common solution to this is automatically grading code based on input/output behavior. Courses often "autograde" programming assignments by running student code on a suite of test inputs and checking whether the returned outputs match those given by the reference implementation. Autograding a programming assignment is usually as simple as running a script on each submission, making it take little to no grader time per student.

Autograding has its disadvantages, though. Firstly, as most interesting programming problems take input from an infinite space (for example, any function which takes an integer as input) it is impossible to test programs exhaustively. Instead, the graders who assemble the test suite are burdened with figuring out a finite set of test inputs which will still detect any incorrect implementation. This involves making blind guesses as to which inputs incorrectly-behaving submissions would have incorrect behavior on, which in turn requires the grader to make blind guesses at how students with incorrect implementations would have structured their code. With so much guesswork involved, it often ends up being the case that students with incorrect code pass all (or almost all) of the test cases the autograder runs their code against.

Secondly, even when graders are lucky enough to assemble a test suite that is exhaustive enough, they are left with the question of how to assign grades based on the output from the autograder. There is no reason to believe that the percentage of test cases a submission passes has bearing on what percentage grade to award it, as even trivial incorrect implementations can pass a high percentage of test cases. Take for example an assignment in which students are asked to implement the merge function from mergesort. Given two sorted lists, the merge function must return a sorted list containing all the elements of the two input lists. An incorrect implementation which simply appends the two lists together would pass any test case where one or both of the input lists is empty, or where all elements of the first list are less than those of the second list. Unless the graders assembling the test suite take special care to consider such trivial incorrect implementations and then reason about the distribution of test cases to minimize the percentage of test cases passed by each trivial incorrect implementation, students who write such implementations would be awarded inapproapriately high grades by a course which assigns grades based on the percentage of test cases passed.

Third, even if a course staff is somehow able to assemble a perfect suite of test cases unaffected by either of these problems, input-output-based autograding is inherently crippled in the level of specificity it can grade to. An autograder would never be able to give different feedback to two students whose implementations have identical input-output behavior but used completely different algorithms to achieve this. This is especially crippling in assignments where students are expected to write code using a specific style or a specific algorithm. In these situations, autograders provide little to no help toward assigning grades to student code submissions. Courses often resort to hand-grading in these situations, which as mentioned before is time-intensive.

Enter Zeus. Zeus allows a course staff to combine the fine-grained specficity and quality of feedback afforded by hand-grading with the speed afforded by using computers to do some of the grading work for you. Zeus operates on the philosophy that for any programming problem in a homework, the vast majority of students are likely to arrive at one of a few common (correct or incorrect) implementations. For each common implementation, rather than separately grading each student who submitted an equivalent to this implementation, a course staff could simply grade each implementation once and assign each student a grade corresponding to that of the implmementation they submitted. Zeus is an efficient algorithm which determines whether two pieces of code are equivalent with respect to grading purposes. It uses this algorithm to arrange student code submissions into equivalence classes, which can then each be graded once.

When graders reason about code, especially when reasoning about aspects of the code such as algorithmic structure, asymptotic complexity, use of helper functions, etc., they tend to think in terms of the structure of the code. Thus they abstract away stylistic details such as choice of variable names, format of pattern-matching, organization of arithmetic, and use of helper abstractions.

For example Figure 1.1 shows two snippets of code which merge two sorted lists together into a new sorted list, a step in the mergesort algorithm. Both approaches have the same philosophy: if either list is empty then return the other list; otherwise if they're both nonempty then return a list whose head is the smaller of the two heads and whose tail is the remainder of the itemrs recursively merged. However the first implementation pattern matches on both lists simultaneously whereas the second pattern matches on them separately; a difference which affects the syntactic structure of the code but doesn't actually affect the algorithm or how we should grade it. Therefore Zeus automatically recognizes the two snippets equivalent.

```
fun merge (l1, l2) =
  case (l1, l2) of
    ([], _) => l2
  | (_::_, []) => l1
  | (x::xs, y::ys) =>
    if x < y
    then x :: merge (xs, l2)
    else y :: merge (l1, ys)
```

```
fun merge (l1, l2) =
  case l1 of
    [] => l2
  | x::xs =>
    case l2 of
      [] => l1
    | y::ys =>
      if x < y
      then x :: merge (xs,
         l2)
      else y :: merge (l1,
         ys)
```

Figure 1.1: Two implementations of merging sorted lists

Figure 1.2 shows two pieces of code which take a list of pairs of numbers and produce a list of each pair's sum. The former directly writes a recursive function, while the second abstracts the recursive nature of the implementation away to the library function `map`. Despite these differences, both implmentations effectively do the same thing and so should be graded the same way. Therefore Zeus indeed automatically recognizes that these implementations are equivalent.

```
fun add_pairs [] = []
  | add_pairs ((a, b)::xs) =
      a+b :: add_pairs xs
```

```
fun map f =
  let
    val rec map_f = fn
      [] => []
    | x::xs => f x :: map_f
       xs
  in
    map_f
  end

val add_pairs = map op+
```

Figure 1.2: Two implementations of adding two optional numbers

Figure 1.3 demonstrates two snippets of code which add two optional numbers together and do so with particularly different style. The first approach directly pattern matches on the optional numbers to perform the logic of adding the numbers together in the case where both are non-null. The second approach on the other hand looks completely different. Rather than directly appealing to the structure of optional values, the second approach only interacts with them as a monadic structure, reflecting a different stylistic philosophy. However as before, underneath this difference in style is the same underlying algorithmic structure, so these pieces of code are equivalent for grading purposes. Appropriately, Zeus automatically detects these pieces of code as the same.

```
fun add_opt x y =
  case (x, y) of
    (SOME m, SOME n) =>
      SOME (m + n)
  | (NONE, _) => NONE
  | (_, NONE) => NONE
```

```
fun bind a f =
  case a of
    SOME b => f b
  | NONE => NONE
val return = SOME

fun add_opt x y =
  bind x (fn m =>
  bind y (fn n =>
    return (m + n)
  ))
```

Figure 1.3: Two implementations of adding two optional numbers

Chapter 2

Related Work

Program equivalence is a fundamental problem in computer science. There are two primary philosophies in which program equivalence is studied.

In theoretical settings, the goal is often to develop relations which precisely capture all situations in which programs are equivalent. These relations are used to reason about equivalence, but tend to be defined in a declarative way which is not conducive to automatic proof search. *Observational equivalence/contextual equivalence* and *extensional equivalence/extensional equivalence* are examples of such relations [5] [2]. Godlin & Strichman [3] present other relations for similar purposes. These are often used in proofs to show that program behavior is preserved by a transformation or algorithm, as in the case of Acar et al. [1] and Stone & Harper [9].

In more applied settings, sometimes we wish to automatically check that two programs are equivalent. An example of this is in compilers; after a compiler applies a transformation it can be useful to have it automatically check that the transformed program is equivalent to the original one. Such approaches are exhibited by Kundu et al. [10] and Necula [8]. Lopes & Monteiro [7] present an algorithm for equivalence of programs with loops and integer arithmetic. As arbitrary program equivalence is undecidable, algorithms for program equivalence (such as ours) are weaker than equivalence relations developed for proofs.

Our algorithm is inspired by the one presented by Stone & Harper [9]. We prove our algorithm sound with respect to extensional equivalence. Hopkins et al. [6] present an algorithm for equivalence of a decidable fragment of ML. Our apporach differs from theirs in that rather than selecting a subset of ML and aiming for completeness, our algorithm is incomplete but operates over the full space of ML programs.

The primary purpose of our algorithm is application to education, as discussed in the previous section. There are other efforts to use notions of program equivalence to aid in working with mass amounts of student code. Gulwani et al. [4] present a similar approach to ours, but applied to Python rather than ML. The notion of equivalence used in their work is suited for simple imperative programs such as those prevalent in introductory courses taught in Python, but does not apply well to higher-order functional programs like those in courses taught in ML.

Chapter 3

LambdaPix

Our algorithm operates over a language which we call *LambdaPix*. In this section we present the syntax and semantics for LambdaPix.

A design goal of LambdaPix is that it be easy to transpile functional programs into. Programs written in ML-based languages, assuming they make limited use of complex features like module systems and algebraic data type declarations, are easy to transpile into LambdaPix. We have implemented transpilation from Standard ML into LambdaPix, and transpilation from languages like OCaml and Haskell should be similarly easy to implement.

LambdaPix is so named because it is the **lambda** calculus enriched with **p**attern matching and **fix**ed points. Arbitrary labeled product types are supported as labeled records. For sum types and recursive types, LambdaPix is defined over an arbitrary fixed set of algebraic data types, with associated injection labels.

Pattern matching plays a big role in LambdaPix, as it is the only mechanism for conditional branching.

3.1 Syntax

Figure 3.1 defines the syntax of LambdaPix.

$$
\begin{array}{lllll}
\textit{pat} & p & ::= & \_ & \textit{wildcard pattern} \\
 & & & x & \textit{variable pattern} \\
 & & & \{\ell_1 = p_1, \ldots, \ell_n = p_n\} & \textit{record pattern} \\
 & & & x \texttt{ as } p & \textit{alias pattern} \\
 & & & c & \textit{constant pattern} \\
 & & & \ell_{i,j} \cdot p & \textit{injection pattern} \\
\textit{exp} & e & ::= & c & \textit{constant} \\
 & & & x & \textit{variable} \\
 & & & \{\ell_1 = e_1, \ldots, \ell_n = e_n\} & \textit{record} \\
 & & & e \cdot \ell_i & \textit{projection} \\
 & & & \ell_{i,j} \cdot e & \textit{injection} \\
 & & & \texttt{case } e \ \{p_1.e_1 \mid \ldots \mid p_n.e_n\} & \textit{case analysis} \\
 & & & \lambda x.e & \textit{abstraction} \\
 & & & e_1 \ e_2 & \textit{application} \\
 & & & \texttt{fix } x \texttt{ is } e & \textit{fixed point}
\end{array}
$$

Figure 3.1: The syntax of LambdaPix

3.2 Static Semantics

The Zeus algorithm is agnostic of a type system, so the following static semantics play no role in how the algorithm operates. We still however present static semantics for LambdaPix as we view Zeus as operating over a statically typed language. Despite the algorithm's type agnosticism, it is presented in terms of these static semantics so as to make it easier to formally reason about.

LambdaPix's static semantics are designed to be nonrestrictive and declarative enough that any reasonable (not necessarily type-directed) transpilation from a well-formed ML-like program would result in a well-formed LambdaPix program. This also helps keep the algorithm language-agnostic.

We assume an arbitrary fixed set of algebraic data types with associated injection labels. In particular, assume a fixed set of judgements of the form $\ell_{i,j} : \tau_{i,j} \hookrightarrow \delta_i$ where i ranges over the fixed set of algebraic data types and j ranges over the injection labels for data type i. We take $\ell_{i,j} : \tau_{i,j} \hookrightarrow \delta_i$ to mean that the type δ_i has a label $\ell_{i,j}$ which accepts arguments of type $\tau_{i,j}$. Note that by allowing $\tau_{i,j}$ to contain instances of δ_i, this data type system affords LamdbaPix a form of inductive types.

Figure 3.2 define the types which LambdaPix expressions may range over. Base types are the types of LambdaPix constants.

$$\begin{array}{llll} \textit{typ} & \tau & ::= & b & \textit{base type} \\ & & & \delta_i & \textit{data type} \\ & & & \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} & \textit{product type} \\ & & & \tau_1 \to \tau_2 & \textit{function type} \end{array}$$

Figure 3.2: The types of LambdaPix expressions

Figure 3.3 defines an auxillary judgement used in the typechecking of expressions: pattern typing. The pattern typing judgement $p :: \tau \dashv \Gamma$ defines that expressions of type τ can be matched against the pattern p, and that doing so produces new variable bindings whose types are captured in Γ. This is used in the typechecking of case expressions.

$$\frac{}{\_ :: \tau \dashv}\ \textsc{PatTy}_1 \qquad \frac{}{x :: \tau \dashv x : \tau}\ \textsc{PatTy}_2$$

$$\frac{p_1 :: \tau_1 \dashv \Gamma_1 \quad \ldots \quad p_n :: \tau_n \dashv \Gamma_n}{\{l_1 = p_1, \ldots, l_n = p_n\} :: \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \dashv \Gamma_1 \ldots \Gamma_n}\ \textsc{PatTy}_3$$

$$\frac{p :: \tau \dashv \Gamma}{x\ \texttt{as}\ p :: \tau \dashv \Gamma, x : \tau}\ \textsc{PatTy}_4 \qquad \frac{}{c :: b \dashv}\ \textsc{PatTy}_5 \qquad \frac{\ell_{i,j} : \tau_{i,j} \hookrightarrow \delta_i \quad p :: \tau_{i,j} \dashv \Gamma}{\ell_{i,j} \cdot p :: \delta_i \dashv \Gamma}\ \textsc{PatTy}_6$$

Figure 3.3: Pattern typing in LambdaPix

Figure 3.4 defines typing for expressions in LambdaPix. A LambdaPix expression e is well-formed if there exists a type τ such that $\cdot \vdash e : \tau$ is derivable from the above typing rules.

Not captured in the type system of LambdaPix are the following two restrictions:

- No variable may appear more than once in a pattern.
- The patterns of a case expression must be exhaustive.

3.3 Dynamic Semantics

Here we define how LambdaPix expressions evaluate. We define evaluation as a small-step dynamic semantics where the judgement $e \Longrightarrow e'$ means that e steps to e' and the judgement e val means that e is a value and doesn't step any further. LambdaPix enjoys progress and preservation: for any typing context Γ and expression e such that $\Gamma \vdash e : \tau$ it is either the case that there exists an expression e' such that $e \Longrightarrow e'$ (in which case $\Gamma \vdash e' : \tau$ as well) or e val. The finality of values is also enjoyed: it is never simultaneously the case that $e \Longrightarrow e'$ and e val.

To define evaluation we first define two helper judgements to deal with pattern matching (Figure 3.5). The judgement $v \mathbin{/\!\!/} p \dashv b$ means the expression v matches to the pattern p

$$\frac{}{\Gamma \vdash c : b}\ \textsc{Ty}_1 \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Ty}_2$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\ell_1 = e_1, \dots, \ell_n = e_n\} : \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\}}\ \textsc{Ty}_3 \qquad \frac{\Gamma \vdash e : \{\dots, \ell_i : \tau_i, \dots\}}{\Gamma \vdash e \cdot \ell_i : \tau_i}\ \textsc{Ty}_4$$

$$\frac{\ell_{i,j} : \tau_{i,j} \hookrightarrow \delta_i \quad \Gamma \vdash e : \tau_{i,j}}{\Gamma \vdash \ell_{i,j} \cdot e : \delta_i}\ \textsc{Ty}_5$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash p_1 :: \tau \dashv \Gamma_1 \quad \Gamma, \Gamma_1 \vdash e_1 : \tau' \quad \dots \quad \Gamma \vdash p_n :: \tau \dashv \Gamma_n \quad \Gamma, \Gamma_n \vdash e_n : \tau'}{\Gamma \vdash \texttt{case}\ e\ \{p_1.e_1 \mid \dots \mid p_n.e_n\} : \tau'}\ \textsc{Ty}_6$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \textsc{Ty}_7 \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{Ty}_8 \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{fix}\ x\ \texttt{is}\ e : \tau}\ \textsc{Ty}_9$$

Figure 3.4: Expression typing in LambdaPix

producing the bindings b, and the judgement $v \not/\!\!/ p$ which means the expression v does not match to the pattern p. It is assumed as a precondition to these judgements that v val, $\vdash v : \tau$, and $p :: \tau$. Pattern matching in LambdaPix enjoys the property that for any v

$$\frac{}{v \mathbin{/\!\!/} \_ \dashv}\ \textsc{Match}_1 \qquad \frac{}{v \mathbin{/\!\!/} x \dashv v/x}\ \textsc{Match}_2$$

$$\frac{v_1 \mathbin{/\!\!/} p_1 \dashv b_1 \quad \dots v_n \mathbin{/\!\!/} p_n \dashv b_n}{\{\ell_1 = v_1, \dots, \ell_n = v_n\} \mathbin{/\!\!/} \{\ell_1 = p_1, \dots, \ell_n = p_n\} \dashv b_1 \dots b_n}\ \textsc{Match}_3$$

$$\frac{v_i \not/\!\!/ p_i}{\{\ell_1 = v_1, \dots, \ell_n = v_n\} \not/\!\!/ \{\ell_1 = p_1, \dots, \ell_n = p_n\}}\ \textsc{Match}_4 \qquad \frac{v \mathbin{/\!\!/} p \dashv b}{v \mathbin{/\!\!/} x\ \texttt{as}\ p \dashv b, v/x}\ \textsc{Match}_5$$

$$\frac{v \not/\!\!/ p}{v \not/\!\!/ x\ \texttt{as}\ p}\ \textsc{Match}_6 \qquad \frac{c_1 = c_2}{c_1 \mathbin{/\!\!/} c_2 \dashv}\ \textsc{Match}_7 \qquad \frac{c_1 \neq c_2}{c_1 \not/\!\!/ c_2}\ \textsc{Match}_8$$

$$\frac{j = k \quad v \mathbin{/\!\!/} p \dashv b}{\ell_{i,j} \cdot v \mathbin{/\!\!/} \ell_{i,k} \cdot p \dashv b}\ \textsc{Match}_9 \qquad \frac{j \neq k}{\ell_{i,j} \cdot v \not/\!\!/ \ell_{i,k} \cdot p}\ \textsc{Match}_{10} \qquad \frac{j = k \quad v \not/\!\!/ p}{\ell_{i,j} \cdot v \not/\!\!/ \ell_{i,k} \cdot p}\ \textsc{Match}_{11}$$

Figure 3.5: Pattern Matching in LambdaPix

and p satisfying the above preconditions it is either the case that there exist bindings b such that $v \mathbin{/\!\!/} p \dashv b$, or $v \not/\!\!/ p$. It is never simultaneously the case that $v \mathbin{/\!\!/} p \dashv b$ and $v \not/\!\!/ p$.

In Figure 3.6 we use these helper judgements to define the evaluation judgements.

The keen reader may observe that rule $\textsc{Dyn}_9$ is nondeterministic as it does not specify which branch is taken when multiple patterns are matched. This rule is presented as it is to make it more straightforward to justify the algorithm's soundness. To resolve this

$$\frac{}{c\ \mathsf{val}}\ \textsc{Dyn}_1 \qquad \frac{\forall 1 \le j < i . e_j\ \mathsf{val} \quad e_i \Longrightarrow e_i'}{\{\ldots, \ell_i = e_i, \ldots\} \Longrightarrow \{\ldots, \ell_i = e_i', \ldots\}}\ \textsc{Dyn}_2$$

$$\frac{\forall 1 \le j \le n . e_j\ \mathsf{val}}{\{\ell_1 = e_1, \ldots, \ell_n = e_n\}\ \mathsf{val}}\ \textsc{Dyn}_3 \qquad \frac{e \Longrightarrow e'}{e \cdot \ell_i \Longrightarrow e' \cdot \ell_i}\ \textsc{Dyn}_4$$

$$\frac{\{\ldots, \ell_i = e_i, \ldots\}\ \mathsf{val}}{\{\ldots, \ell_i = e_i, \ldots\} \cdot \ell_i \Longrightarrow e_i}\ \textsc{Dyn}_5 \qquad \frac{e \Longrightarrow e'}{\ell_{i,j} \cdot e \Longrightarrow \ell_{i,j} \cdot e'}\ \textsc{Dyn}_6 \qquad \frac{e\ \mathsf{val}}{\ell_{i,j} \cdot e\ \mathsf{val}}\ \textsc{Dyn}_7$$

$$\frac{e \Longrightarrow e'}{\mathtt{case}\ e\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\} \Longrightarrow \mathtt{case}\ e'\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\}}\ \textsc{Dyn}_8$$

$$\frac{e\ \mathsf{val} \quad e \mathbin{/\!\!/} p_i \dashv b}{\mathtt{case}\ e\ \{\ldots \mid p_i.e_i \mid \ldots\} \Longrightarrow [b]e_i}\ \textsc{Dyn}_9 \qquad \frac{}{\lambda x.e\ \mathsf{val}}\ \textsc{Dyn}_{10} \qquad \frac{e_1 \Longrightarrow e_1'}{e_1\ e_2 \Longrightarrow e_1'\ e_2}\ \textsc{Dyn}_{11}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \Longrightarrow e_2'}{e_1\ e_2 \Longrightarrow e_1\ e_2'}\ \textsc{Dyn}_{12} \qquad \frac{e_2\ \mathsf{val}}{(\lambda x.e)\ e_2 \Longrightarrow [e_2/x]e}\ \textsc{Dyn}_{13}$$

$$\frac{}{\mathtt{fix}\ x\ \mathtt{is}\ e \Longrightarrow [\mathtt{fix}\ x\ \mathtt{is}\ e/x]e}\ \textsc{Dyn}_{14}$$

Figure 3.6: Dynamic semantics of LambdaPix

nondeterminism, assume that the branch corresponding to the first matched pattern is the one taken by the dynamics.

We use these judgements to define what it means for an expression to evaluate to a value. We use $e \Downarrow v$ to denote that e evaluates to v. In Figure 3.7 big-step dynamics are defined as the transitive closure of the small-step dynamics:

$$\frac{v\ \mathsf{val}}{v \Downarrow v}\ \textsc{BigDyn}_1 \qquad \frac{e \Longrightarrow e' \quad e \Downarrow v}{e \Downarrow v}\ \textsc{BigDyn}_2$$

Figure 3.7: Big-Step Dynamic Semantics of LambdaPix

Chapter 4

The Algorithm

The algorithm takes as input two LambdaPix expressions of the same type and outputs a boolean value indicating whether they have been found to be equal. It generates a propositional logic formula which is valid only if the two expressions are equivalent, then makes use of an SMT solver to check whether the formula is valid.

4.1 Propositional Logic Formulas

Figure 4.1 defines the form of the formulas generated by the algorithm. The leaves of these formulas are equalities between LambdaPix terms. A structural equality $e_1 = e_2$ is true if e_1 and e_2 are identical and false otherwise. The only two connectives are conjunction and implication. A conjunction $\sigma_1 \wedge \sigma_2$ is true if both σ_1 and σ_2 are true, and false otherwise. An implication $\sigma_1 \Rightarrow \sigma_2$ is true either if σ_1 is false or if σ_2 is true.

$$\begin{array}{rrrl} \sigma & ::= & e_1 = e_2 & \textit{structural expression equality} \\ & & \sigma_1 \wedge \sigma_2 & \textit{conjunction} \\ & & \sigma_1 \Rightarrow \sigma_2 & \textit{implication} \end{array}$$

Figure 4.1: Propositional Logic Formulas

Validity $\overset{\mathsf{val}}{\forall}_\Gamma . \sigma$

Formulas may contain free variables which must be resolved in order to gauge the formula's truth value. For our purposes, free variables will only appear inside expressions used in structural equality formulas. We therefore define variable substitution into formulas as substituting into these expressions; for example, $[3/x](x = 1 \wedge 2 = x)$ yields $3 = 1 \wedge 2 = 3$.

A formula is *valid* if it is true under all possible substitutions of its variables. To denote this, we first define a new form of judgement: If $\Gamma = \vec{x} : \vec{\tau}$, then the judgement $\overset{\mathsf{val}}{\forall}_\Gamma . j$ holds if for all $\vec{v}$ where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$, it is the case that $[\vec{v}/\vec{x}]j$ holds. Then if Γ is a typing context with a mapping for every free variable in a formula σ, the validity of σ is denoted $\overset{\mathsf{val}}{\forall}_{\Gamma}.\sigma$.

4.2 Formula Generation

The most important judgement for formula generation is $\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$. Two expressions e_1 and e_2 where $\cdot \vdash e_1 : \tau$ and $\cdot \vdash e_2 : \tau$ are isomorphic if $\cdot \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ and $\forall_{\Gamma'}.\sigma$.

To define this judgement we begin by defining a few helper judgements.

Weak Head Normal Reduction $e \downarrow e'$

It would be wasteful to fully evaluate each expression at each recursive iteration of the algorithm, as at any point we look no deeper than the head symbol. Additionally, we don't even have the option of fully evaluating the expressions during the execution of the algorithm, as expressions may contain free variables in redex positions. For this reason we instead use weak head normal reduction at each step; this eliminates head-position redexes until free variables get in the way. It does not evaluate subexpressions any more than necessary. This approach is inspired by Stone & Harper [9].

$$\frac{e \rightsquigarrow e' \quad e' \downarrow e''}{e \downarrow e''}\ \textsc{BigWhnf}_1 \qquad \frac{e \not\rightsquigarrow}{e \downarrow e}\ \textsc{BigWhnf}_2$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2}\ \textsc{Whnf}_1 \qquad \frac{}{(\lambda x.e_1)e_2 \rightsquigarrow [e_2/x]e_1}\ \textsc{Whnf}_2 \qquad \frac{e \rightsquigarrow e'}{e \cdot \ell_i \rightsquigarrow e' \cdot \ell_i}\ \textsc{Whnf}_3$$

$$\frac{}{\{\dots, \ell_i = e, \dots\} \cdot \ell_i \rightsquigarrow e}\ \textsc{Whnf}_4$$

Atomic Expressions e atomic

The algorithm can be thought of as breaking down expressions until they are "small enough" to be used in a structural equality formula. The judgement e atomic defines that e is small enough for this. An expression is atomic if it does not contain complicated constructs like functions, applications, fixed points, or case expressions.

$$\frac{}{c\ \mathsf{atomic}}\ \textsc{Atomic}_1 \qquad \frac{}{x\ \mathsf{atomic}}\ \textsc{Atomic}_2 \qquad \frac{e_1\ \mathsf{atomic} \quad \dots \quad e_n\ \mathsf{atomic}}{\{\ell_1 = e_1, \dots, \ell_n = e_n\}\ \mathsf{atomic}}\ \textsc{Atomic}_3$$

$$\frac{e\ \mathsf{atomic}}{e \cdot \ell_i\ \mathsf{atomic}}\ \textsc{Atomic}_4 \qquad \frac{e\ \mathsf{atomic}}{\ell_i \cdot e\ \mathsf{atomic}}\ \textsc{Atomic}_5$$

Freshening $\mathsf{freshen}\ p.e \hookrightarrow p'.e'$

It is sometimes useful to generate fresh variables to avoid variable capture. As single variables aren't the only form of binding sites in LamdbaPix, we generalize this notion to patterns.

When $\mathsf{freshen}\ p.e \hookrightarrow p'.e'$, $p'.e'$ is the same as $p.e$ except with all variables bound by p having been alpha-varied to fresh variables.

$$\frac{}{\mathsf{freshen}\ \_.e \hookrightarrow \_.e}\ \textsc{Freshen}_1 \qquad \frac{y\ \mathsf{fresh}}{\mathsf{freshen}\ x.e \hookrightarrow y.[y/x]e}\ \textsc{Freshen}_2$$

$$\frac{\mathsf{freshen}\ p_1.e \hookrightarrow p'_1.e_1 \quad \ldots \quad \mathsf{freshen}\ p_n.e_{n-1} \hookrightarrow p'_n.e_n}{\mathsf{freshen}\ \{\ell_1 = p_1, \ldots, \ell_n = p_n\}.e \hookrightarrow \{\ell_1 = p'_1, \ldots, \ell_n = p'_n\}.e_n}\ \textsc{Freshen}_3$$

$$\frac{y\ \mathsf{fresh} \quad \mathsf{freshen}\ p.e \hookrightarrow p'.e'}{\mathsf{freshen}\ x\ \mathsf{as}\ p.e \hookrightarrow y\ \mathsf{as}\ p'.[y/x]e'}\ \textsc{Freshen}_4 \qquad \frac{}{\mathsf{freshen}\ c.e \hookrightarrow c.e}\ \textsc{Freshen}_5$$

$$\frac{\mathsf{freshen}\ p.e \hookrightarrow p'.e'}{\mathsf{freshen}\ \ell_i \cdot p.e \hookrightarrow \ell_i \cdot p'.e'}\ \textsc{Freshen}_6$$

Formula Generation for Expressions $\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$

This judgement is mutually recursive with $\Gamma \vdash e_1 \leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$.

When $\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$, the only free variables appearing in e_1 and e_2 are in Γ so $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. σ on the other hand can contain more free variables than just those in Γ. The purpose of Γ' is to describe the rest of the variables in σ. Γ and Γ' are disjoint and between them account for all variables which may appear in σ.

$$\frac{e_1 \downarrow e'_1 \quad e_2 \downarrow e'_2 \quad \Gamma \vdash e'_1 \leftrightarrow e'_2 : \tau \dashv \Gamma'; \sigma}{\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma}\ \textsc{IsoExp}$$

Formula Generation for WHNF Expressions $\Gamma \vdash e_1 \leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$

This judgement does the bulk of the work for the algorithm. It assumes as a precondition that e_1 and e_2 are in weak head normal form. As with formula generation for arbitrary expressions, $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$, Γ and Γ' are disjoint, and all the free variables in σ are captured within Γ and Γ'.

$$\frac{e_1 \ \textsf{atomic} \quad e_2 \ \textsf{atomic}}{\Gamma \vdash e_1 \leftrightarrow e_2 : \tau \dashv \cdot ; e_1 = e_2} \ \textsc{Iso}_1$$

$$\frac{\Gamma \vdash e_1 \Leftrightarrow e_1' : \tau_1 \dashv \Gamma_1' ; \sigma_1 \quad \ldots \quad \Gamma \vdash e_n \Leftrightarrow e_n' : \tau_n \dashv \Gamma_n' ; \sigma_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \leftrightarrow \{\ell_1 = e_1', \ldots, \ell_n = e_n'\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \dashv \Gamma_1', \ldots, \Gamma_n' ; \sigma_1 \wedge \ldots \wedge \sigma_n} \ \textsc{Iso}_2$$

$$\frac{\Gamma \vdash e_1 \leftrightarrow e_2 : \{\ldots, \ell_i : \tau_i, \ldots\} \dashv \Gamma' ; \sigma}{\Gamma \vdash e_1 \cdot \ell_i \leftrightarrow e_2 \cdot \ell_i : \tau_i \dashv \Gamma' ; \sigma} \ \textsc{Iso}_3 \qquad \frac{\ell_j : \tau_{i,j} \hookrightarrow \delta_i \quad \Gamma \vdash e_1 \Leftrightarrow e_2 : \tau_{i,j} \dashv \Gamma' ; \sigma}{\Gamma \vdash \ell_j \cdot e_1 \leftrightarrow \ell_j \cdot e_2 : \delta_i \dashv \Gamma' ; \sigma} \ \textsc{Iso}_4$$

$$\frac{e \ \textsf{atomic} \quad \forall_{i \in [n]} \, (\textsf{freshen} \ p_i.e_i \hookrightarrow p_i'.e_i' \quad p_i' :: \tau' \dashv \Gamma_i \quad \Gamma, \Gamma_i \vdash e_i' \Leftrightarrow e' : \tau \dashv \Gamma_i' ; \sigma_i)}{\Gamma \vdash \texttt{case} \ e \ \{p_1.e_1 \mid \ldots \mid p_n.e_n\} \leftrightarrow e' : \tau \dashv \Gamma_1, \Gamma_1', \ldots, \Gamma_n, \Gamma_n' ; \wedge_{i \in [n]} \, (e = p_i' \Rightarrow \sigma_i)} \ \textsc{Iso}_5$$

$$\frac{e \ \textsf{atomic} \quad \forall_{i \in [n]} \, (\textsf{freshen} \ p_i.e_i \hookrightarrow p_i'.e_i' \quad p_i' :: \tau' \dashv \Gamma_i \quad \Gamma, \Gamma_i \vdash e' \Leftrightarrow e_i' : \tau \dashv \Gamma_i' ; \sigma_1)}{\Gamma \vdash e' \leftrightarrow \texttt{case} \ e \ \{p_1.e_1 \mid \ldots \mid p_n.e_n\} : \tau \dashv \Gamma_1, \Gamma_1', \ldots, \Gamma_n, \Gamma_n' ; \wedge_{i \in [n]} \, (e = p_i' \Rightarrow \sigma_i)} \ \textsc{Iso}_6$$

$$\frac{\Gamma \vdash e \leftrightarrow e' : \tau' \dashv \Gamma' ; \sigma \quad x \ \textsf{fresh} \quad \Gamma, x : \tau' \vdash \texttt{case} \ x \ \{\ldots\} \leftrightarrow \texttt{case} \ x \ \{\ldots'\} : \tau \dashv \Gamma'' ; \sigma'}{\Gamma \vdash \texttt{case} \ e \ \{\ldots\} \leftrightarrow \texttt{case} \ e' \ \{\ldots'\} : \tau \dashv \Gamma', x : \tau', \Gamma'' ; \sigma \wedge \sigma'} \ \textsc{Iso}_7$$

$$\frac{x \ \textsf{fresh} \quad \Gamma, x : \tau \vdash [x/x_1]e_1 \Leftrightarrow [x/x_2]e_2 : \tau' \dashv \Gamma' ; \sigma}{\Gamma \vdash \lambda x_1.e_1 \leftrightarrow \lambda x_2.e_2 : \tau \to \tau' \dashv x : \tau, \Gamma' ; \sigma} \ \textsc{Iso}_8$$

$$\frac{\Gamma \vdash e_1 \leftrightarrow e_2 : \tau \to \tau' \dashv \Gamma' ; \sigma \quad \Gamma \vdash e_1' \Leftrightarrow e_2' : \tau \dashv \Gamma'' ; \sigma'}{\Gamma \vdash e_1 \ e_1' \leftrightarrow e_2 \ e_2' : \tau' \dashv \Gamma', \Gamma'' ; \sigma \wedge \sigma'} \ \textsc{Iso}_9$$

$$\frac{x \ \textsf{fresh} \quad \Gamma, x : \tau \vdash [x/x_1]e_1 \Leftrightarrow [x/x_2]e_2 : \tau \dashv \Gamma' ; \sigma}{\Gamma \vdash \texttt{fix} \ x_1 \ \texttt{is} \ e_1 \leftrightarrow \texttt{fix} \ x_2 \ \texttt{is} \ e_2 : \tau \dashv x : \tau, \Gamma' ; \sigma} \ \textsc{Iso}_{10}$$

Chapter 5

Operation

In this section, we understand the algorithm by looking at how it operates on the two programs in Figure 1.3. As we step through this example, we will refer to the inference rules from the previous section and watch how they are applied. First, we transpile both programs to LambdaPix. This is shown in Figure 5.1.

```
λ x . λ y .
  case (x, y) of
    (SOME·m, SOME·n) . SOME·(m
        + n)
  | (NONE·(), _) . NONE·()
  | (_, NONE·()) . NONE·()
```

```
(λ return . λ bind .
  λ x . λ y .
    bind x (λ m .
    bind y (λ n .
      return (m + n)
    ))
)
(λ e . SOME·e)
(λ a . λ f .
  case a of
    SOME·b . f b
  | NONE·() . NONE·()
)
```

Figure 5.1: Figure 1.3, translated to LambdaPix

Since much of the proof derivation which drives the algorithm is free of branching, through most of this section we will view the algorithm as transforming the above programs through the application of rules, rather than building up a proof tree.

The entry point to the algorithm is the $\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ judgement, defined by rule IsoExp. By this rule, we reduce both programs to weak head normal form then apply the $\Gamma \vdash e_1 \leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ judgement to them. The first program is already in weak head normal form. To get the second program into weak head normal form, we apply the beta reduction rules Whnf$_1$ and Whnf$_2$ a few times. After weak head normal reducing the second program, we are left with the pair of programs in Figure 5.2.

Next since both programs are lambda expressions with two curried arguments, we proceed with two applications of the extensionality rule Iso$_8$. This constitutes making up a

```
λ x . λ y .
  case (x, y) of
    (SOME·m, SOME·n) . SOME·(m
        + n)
  | (NONE·(), _) . NONE·()
  | (_, NONE·()) . NONE·()
```

```
λ x . λ y .
  (λ a . λ f .
    case a of
      SOME·b . f b
    | NONE·() . NONE·()
  ) x (λ m .
  (λ a . λ f .
    case a of
      SOME·b . f b
    | NONE·() . NONE·()
  ) y (λ n .
    (λ e . SOME·e) (m + n)
  ))
```

Figure 5.2: Figure 5.1 after reducing to WHNF

fresh variable for the function argument and substituting this same fresh variable into both programs for the function argument. Since these variables are named the same in both programs, for the sake of this demonstration we'll simply "substitute" the variables named x with x and the ones named y with y. These applications of Iso$_8$ gets us to Figure 5.3.

```
case (x, y) of
  (SOME·m, SOME·n) . SOME·(m +
      n)
| (NONE·(), _) . NONE·()
| (_, NONE·()) . NONE·()
```

```
(λ a . λ f .
  case a of
    SOME·b . f b
  | NONE·() . NONE·()
) x (λ m .
(λ a . λ f .
  case a of
    SOME·b . f b
  | NONE·() . NONE·()
) y (λ n .
  (λ e . SOME·e) (m + n)
))
```

Figure 5.3: Figure 5.2 after applying Iso$_8$ twice

Since the premise of Iso$_8$ invokes the $\Leftrightarrow$ judgement, we again reduce to WHNF. This constitutes more beta reductions. This next step is shown in Figure 5.4.

Since both programs are case expressions, we now need to choose between rules Iso$_5$, Iso$_6$, and Iso$_7$. We always prefer Iso$_5$ and Iso$_6$ over Iso$_7$ because the "asymmetric" case rules are more intellgent than the "symmetric" one. However, the asymmetric rules can't always be applied; they only can be when the subject of the case expression is atomic. Fortunately, by rules Atomic$_2$ and Atomic$_3$, the subjects of these case expressions are atomic, so we can

```
case (x, y) of
  (SOME·m, SOME·n) . SOME·(m +
      n)
| (NONE·(), _) . NONE·()
| (_, NONE·()) . NONE·()
```

```
case x of
  SOME·b .
    (λ m .
    (λ a . λ f .
      case a of
        SOME·b . f b
      | NONE·() . NONE·()
    ) y (λ n .
      (λ e . SOME·e) (m + n)
    )) b
| NONE·() . NONE·()
```

Figure 5.4: Figure 5.3 after reducing to WHNF

use $\textsc{Iso}_5$ and $\textsc{Iso}_6$. The order in which they are applied make no difference, so we arbitrarily apply $\textsc{Iso}_5$ first. As there are three branches in the first program's case expression, the algorithm now splits into three branches. To get a feel for the algorithm we'll explicitly step through just the first of these branches, as the other ones work in similar ways.

We "freshen" the branch selected to avoid variable capture. In this situation we'll freshen the first branch of the case expression in the first program by replacing `m` and `n` with `m1` and `n1`, respectively. From this branch we'll generate a formula of the form

$$\big(\texttt{(x, y)} = \texttt{(SOME·m1, SOME·n1)}\big) \Rightarrow \ldots$$

where the ... is what we are going to fill in as we complete this branch of the algorithm. We are now at the point shown in Figure 5.5.

```
SOME·(m1 + n1)
```

```
case x of
  SOME·b .
    (λ m .
    (λ a . λ f .
      case a of
        SOME·b . f b
      | NONE·() . NONE·()
    ) y (λ n .
      (λ e . SOME·e) (m + n)
    )) b
| NONE·() . NONE·()
```

Figure 5.5: The first branch of the algorithm after applying $\textsc{Iso}_5$ to Figure 5.4

We next do a similar step by applying rule $\textsc{Iso}_6$, which branches the algorithm into two. We again focus on the first branch. $\textsc{Iso}_6$ again calls for the branch to be freshened, which we will demonstrate by replacing `b` with `b1`. This branch will therefore generate a formula of the form

$$(\texttt{x} = \texttt{SOME}\cdot\texttt{b1}) \Rightarrow \ldots$$

where again, the rest of this branch will fill in the .... This brings us to Figure 5.6.

```
                                  (λ m .
                                  (λ a . λ f .
                                    case a of
                                      SOME·b . f b
SOME·(m1 + n1)                      | NONE·() . NONE·()
                                  ) y (λ n .
                                    (λ e . SOME·e) (m + n)
                                  )) b1
```

Figure 5.6: The first branch of the algorithm after applying Iso$_6$ to Figure 5.5

We next reduce to WHNF again, which beta reduces the second program: Figure 5.7.

```
                                  case y of
                                    SOME·b .
SOME·(m1 + n1)                        (λ n .
                                        (λ e . SOME·e) (b1 + n)
                                      ) b
                                  | NONE·() . NONE·()
```

Figure 5.7: Figure 5.6 converted to WHNF

Next we apply Iso$_6$ again. We'll freshen by replacing `b` with `b2` [1]. This again branches the proof into two. The first branch corresponds to

$$(\texttt{y} = \texttt{SOME}\cdot\texttt{b2}) \Rightarrow \ldots$$

where we'll fill in the ... next. Taking the first branch brings us to Figure 5.8.

```
                                  (λ n .
SOME·(m1 + n1)                      (λ e . SOME·e) (b1 + n)
                                  ) b2
```

Figure 5.8: The first branch after applying Iso$_6$ to Figure 5.7

We convert to WHNF again, performing beta reductions, bringing us to Figure 5.9.

[1]This demonstrates the necessity of freshening; without it the variable `b` would have two different meanings.

```
SOME·(m1 + n1)                    SOME·(b1 + b2)
```

Figure 5.9: Figure 5.8 after converting to WHNF

As these expressions are both atomic, we complete this branch of the algorithm by generating the formula

$$\texttt{SOME}\cdot\texttt{(m1+n1)} = \texttt{SOME}\cdot\texttt{(b1+b2)}$$

.

Putting together all the branches we've looked at, our overall formula will look something like:

$$
\begin{array}{l}
(((\texttt{x, y}) = (\texttt{SOME}\cdot\texttt{m1, SOME}\cdot\texttt{n1})) \Rightarrow \\
\quad ((\texttt{x} = \texttt{SOME}\cdot\texttt{b1}) \Rightarrow \\
\quad\quad (\texttt{y} = \texttt{SOME}\cdot\texttt{b2}) \Rightarrow (\texttt{SOME}\cdot\texttt{(m1+n1)} = \texttt{SOME}\cdot\texttt{(b1+b2)})\wedge \\
\quad\quad \ldots \\
\quad)\wedge \\
\quad \ldots \\
)\wedge \\
\ldots
\end{array}
$$

where the ... now correspond to the branches of the algorithm we didn't explicitly step into.

Adding in these other branches, we get the full formula:

$$
\begin{array}{l}
(((\texttt{x, y}) = (\texttt{SOME}\cdot\texttt{m1, SOME}\cdot\texttt{n1})) \Rightarrow \\
\quad ((\texttt{x} = \texttt{SOME}\cdot\texttt{b1}) \Rightarrow \\
\quad\quad (\texttt{y} = \texttt{SOME}\cdot\texttt{b2}) \Rightarrow (\texttt{SOME}\cdot\texttt{(m1+n1)} = \texttt{SOME}\cdot\texttt{(b1+b2)}) \wedge \\
\quad\quad (\texttt{y} = \texttt{NONE}\cdot\texttt{()}) \Rightarrow (\texttt{SOME}\cdot\texttt{(m1+n1)} = \texttt{NONE}\cdot\texttt{()}) \\
\quad) \wedge \\
\quad ((\texttt{x} = \texttt{NONE}\cdot\texttt{()}) \Rightarrow \\
\quad\quad (\texttt{y} = \texttt{SOME}\cdot\texttt{b2}) \Rightarrow (\texttt{SOME}\cdot\texttt{(m1+n1)} = \texttt{NONE}\cdot\texttt{()}) \wedge \\
\quad\quad (\texttt{y} = \texttt{NONE}\cdot\texttt{()}) \Rightarrow (\texttt{SOME}\cdot\texttt{(m1+n1)} = \texttt{NONE}\cdot\texttt{()}) \\
\quad) \\
) \wedge \\
(((\texttt{x, y}) = (\texttt{NONE}\cdot\texttt{(), \_})) \Rightarrow \\
\quad ((\texttt{x} = \texttt{SOME}\cdot\texttt{b1}) \Rightarrow \\
\quad\quad (\texttt{y} = \texttt{SOME}\cdot\texttt{b2}) \Rightarrow (\texttt{NONE}\cdot\texttt{()} = \texttt{SOME}\cdot\texttt{(b1+b2)}) \wedge \\
\quad\quad (\texttt{y} = \texttt{NONE}\cdot\texttt{()}) \Rightarrow (\texttt{NONE}\cdot\texttt{()} = \texttt{NONE}\cdot\texttt{()}) \\
\quad) \wedge \\
\quad ((\texttt{x} = \texttt{NONE}\cdot\texttt{()}) \Rightarrow \\
\quad\quad (\texttt{y} = \texttt{SOME}\cdot\texttt{b2}) \Rightarrow (\texttt{NONE}\cdot\texttt{()} = \texttt{NONE}\cdot\texttt{()}) \wedge \\
\quad\quad (\texttt{y} = \texttt{NONE}\cdot\texttt{()}) \Rightarrow (\texttt{NONE}\cdot\texttt{()} = \texttt{NONE}\cdot\texttt{()}) \\
\quad) \\
) \wedge \\
(((\texttt{x, y}) = (\texttt{\_, NONE}\cdot\texttt{()})) \Rightarrow \\
\quad ((\texttt{x} = \texttt{SOME}\cdot\texttt{b1}) \Rightarrow \\
\quad\quad (\texttt{y} = \texttt{SOME}\cdot\texttt{b2}) \Rightarrow (\texttt{NONE}\cdot\texttt{()} = \texttt{SOME}\cdot\texttt{(b1+b2)}) \wedge \\
\quad\quad (\texttt{y} = \texttt{NONE}\cdot\texttt{()}) \Rightarrow (\texttt{NONE}\cdot\texttt{()} = \texttt{NONE}\cdot\texttt{()}) \\
\quad) \wedge \\
\quad ((\texttt{x} = \texttt{NONE}\cdot\texttt{()}) \Rightarrow \\
\quad\quad (\texttt{y} = \texttt{SOME}\cdot\texttt{b2}) \Rightarrow (\texttt{NONE}\cdot\texttt{()} = \texttt{NONE}\cdot\texttt{()}) \wedge \\
\quad\quad (\texttt{y} = \texttt{NONE}\cdot\texttt{()}) \Rightarrow (\texttt{NONE}\cdot\texttt{()} = \texttt{NONE}\cdot\texttt{()}) \\
\quad) \\
)
\end{array}
$$

This formula is then fed into an SMT solver to determine whether it's valid. In this situation, since the two programs are indeed equivalent, this formula is valid.

Chapter 6

Soundness

The Zeus algorithm is not complete, but it is sound. This means that if two programs are equivalent, it's possible that Zeus would not recognize them as equivalent; however, if Zeus recognizes two programs as equivalent then they must *actually* be equivalent.

6.1 Extensional Equivalence

To prove the soundness of Zeus, we must first define what it means for two programs to actually be equivalent. For this we introduce *extensional equivalence*, a widely accepted notion of equivalence. Prior work has shown that extensional equivalence is the same as contextual equivalence, and so two programs which are extensionally equivalent are effectively indistinguishable in terms of behavior. Extensional equivalence is also an equivalence relation, so we may assume that it is reflexive, symmetric, and transitive. LambdaPix enjoys referential transparency, meaning that extensional equivalence of LambdaPix expressions is closed under replacement of subexpressions with extensionally equivalent subexpressions.

We use $e_1 \cong e_2 : \tau$ to denote that expressions e_1 and e_2 are extensionally equivalent and both have the type τ. Unlike our algorithm, extensional equivalence inducts over the types of the expressions rather than their syntax, and is defined only over closed expressions. A precondition to the judgement $e_1 \cong e_2 : \tau$ is that $\vdash e_1 : \tau$ and $\vdash e_2 : \tau$.

As we are only concerned with proving our algorithm sound over valuable expressions, we leave extensional equivalence undefined for divergent expressions. We define that $e_1 \cong e_2 : \tau$ if $\cdot \vdash e_1 : \tau$, $\cdot \vdash e_2 : \tau$, $e_1 \Downarrow v_1$, $e_2 \Downarrow v_2$, and

1. Rule EQ$_1$: In the case that $\tau = \tau_1 \to \tau_2$, for all expressions v such that $\cdot \vdash v : \tau_1$, $v_1\ v \cong v_2\ v : \tau_2$.

2. Rule EQ$_2$: In the case that τ is not an arrow type, for all patterns p such that $p :: \tau$, either $v_1 /\!\!/ p \dashv b$ and $v_2 /\!\!/ p \dashv b$ or $v_1 \not/\!\!/ p$ and $v_2 \not/\!\!/ p$.

This is an atypical formalization of extensional equivalence; it is typically defined in terms of the elimination forms of each type connective. However, since pattern matching in LambdaPix subsumes the elimination of all connectives other than arrows, we simply define equivalence at all non-arrow types in terms of pattern matching.

6.2 Proof

Here we prove that Zeus is sound. This proof makes use of a few lemmas:

- Lemma 1: if $\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ or $\Gamma \vdash e_1 \leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$, then Γ and Γ' are disjoint. *Proof*: by induction on $\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ and $\Gamma \vdash e_1 \leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$

- Lemma 2: WHNF reduction preserves equivalence:
  if $\Gamma \vdash e_1' \cong e_2'$, $e_1 \downarrow e_1'$, and $e_2 \downarrow e_2'$, then $\Gamma \vdash e_1 \cong e_2$. *Proof*: By induction on $e \rightsquigarrow e'$ and appealing to the dynamics, we have that if $\Gamma \vdash e_1' \cong e_2'$, $e_1 \rightsquigarrow e_1'$, and $e_1 \rightsquigarrow e_2'$ then $\Gamma \vdash e_1 \cong e_2$. The rest goes through by induction on $e \downarrow e'$.

- Lemma 3: If $\Gamma \vdash v : \tau$, $p :: \tau \dashv \Gamma'$, $\Gamma, \Gamma' \vdash e : \tau'$, $v \mathbin{/\!/} p \dashv b$, and $\mathsf{freshen}\ p.e \hookrightarrow p'.e'$, then $v \mathbin{/\!/} p' \dashv b'$ and $[b]e = [b']e'$. *Proof*: by induction on $\mathsf{freshen}\ p.e \hookrightarrow p'.e'$.

- Lemma 4: If $v \mathbin{/\!/} p \dashv b$ then $v = [b]p$. *Proof*: by induction on $v \mathbin{/\!/} p \dashv b$.

- Lemma 5: If $e_1 \cong e_2 : \tau$, $e_1 \Downarrow v_1$, and $e_2 \Downarrow v_2$, then for all patterns p where $p :: \tau \dashv \Gamma$, it is the case that either $v_1 \mathbin{/\!/} p \dashv b$ and $v_1 \mathbin{/\!/} p \dashv b$ or $v_1 \not\mathbin{/\!/} p$ and $v_2 \not\mathbin{/\!/} p$. *Proof*: by induction on $e_1 \cong e_2 : \tau$. If $\tau = \tau_1 \to \tau_2$ then by inversion of $p :: \tau \dashv \Gamma$, p must either be a wildcard or a variable. Then by $\textsc{Match}_1$ and $\textsc{Match}_2$, we have that $v_1 \mathbin{/\!/} p \dashv b$ and $v_2 \mathbin{/\!/} p \dashv b$. If τ is not an arrow type, then we conclude by EQ_2.

- Lemma 6: If e atomic then $\vdash e : \tau$ where τ is not an arrow type. *Proof*: by induction on e atomic and appealing to the statics.

- Lemma 7: If $\Gamma \vdash e : \tau$, e atomic, and e is in weak head normal form, then $\overset{\mathsf{val}}{\forall}_\Gamma . e\ \mathsf{val}$. *Proof*: by induction on e atomic and appealing to the dynamics.

We state the theorem of soundness which we will prove:

Theorem For any expressions e_1 and e_2,
if $\cdot \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ and $\overset{\mathsf{val}}{\forall}_{\Gamma'} . \sigma$ then $e_1 \cong e_2 : \tau$.

Proof As the $\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ judgement is defined simultaneously with the $\Gamma \vdash e_1 \leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ judgement, we prove the theorem by simultaneous induction on both of these judgements. We also use the $\overset{\mathsf{val}}{\forall}_\Gamma . j$ judgement to strengthen the inductive hypotheses to account for variables. Recall that if $\Gamma = \vec{x} : \vec{\tau}$, then the judgement $\overset{\mathsf{val}}{\forall}_\Gamma . j$ holds if for all $\vec{v}$ where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$, it is the case that $[\vec{v}/\vec{x}]j$ holds. The theorem we wish to show by induction is then:

- If $\Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ then $\overset{\mathsf{val}}{\forall}_\Gamma . \left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'} . \sigma\right) \text{ then } e_1 \cong e_2 : \tau\right)$.

- If $\Gamma \vdash e_1 \leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ then $\overset{\mathsf{val}}{\forall}_\Gamma . \left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'} . \sigma\right) \text{ then } e_1 \cong e_2 : \tau\right)$.

We first verify that this is sufficient to show the theorem. Indeed, when $\cdot \vdash e_1 \Leftrightarrow e_2 : \tau \dashv \Gamma'; \sigma$ we have $\overset{\mathsf{val}}{\forall}_{\cdot}.\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } e_1 \cong e_2 : \tau\right)$. As the outer quantifier quantifies over no variables, we have that if $\left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right)$ then $e_1 \cong e_2 : \tau$. This together with the assumption that $\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma$ allows us to conclude that $e_1 \cong e_2 : \tau$.

We proceed to prove each rule. In cases where the exact type of the expressions are obvious or irrelevant, we use the shorthand $e \cong e'$ to mean that $e \cong e' : \tau$ for some type τ.

- IsoExp: Let $\Gamma = \vec{x} : \vec{\tau}$ and let $\vec{v}$ be arbitrary where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$. Assume $[\vec{v}/\vec{x}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right)$.

  It must be shown that $[\vec{v}/\vec{x}](e_1 \cong e_2)$.

  By the inductive hypothesis we have that $\overset{\mathsf{val}}{\forall}_{\Gamma}.\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } e'_1 \cong e'_2\right)$, and therefore
  $$[\vec{v}/\vec{x}]\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } e'_1 \cong e'_2\right)$$
  equivalently,
  $$\text{if } [\vec{v}/\vec{x}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } [\vec{v}/\vec{x}](e'_1 \cong e'_2)$$
  As we have $[\vec{v}/\vec{x}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right)$ by assumption, we may conclude
  $$[\vec{v}/\vec{x}](e'_1 \cong e'_2)$$

- Iso$_1$: Let $\Gamma = \vec{x} : \vec{\tau}$ and let $\vec{v}$ be arbitrary where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$. Assume $[\vec{v}/\vec{x}]\left(\overset{\mathsf{val}}{\forall}_{\cdot}.e_1 = e_2\right)$, or equivalently $[\vec{v}/\vec{x}](e_1 = e_2)$.

  It must be shown that $[\vec{v}/\vec{x}](e_1 \cong e_2)$.

  As $[\vec{v}/\vec{x}](e_1 = e_2)$, we have $[\vec{v}/\vec{x}](e_1 \cong e_2)$ by reflexivity.

- Iso$_2$: Let $\Gamma = \vec{x} : \vec{\tau}$ and let $\vec{v}$ be arbitrary where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$. Assume $[\vec{v}/\vec{x}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma'_1,\ldots,\Gamma'_n}.\sigma_1 \wedge \ldots \sigma_n\right)$.

  It must be shown that $[\vec{v}/\vec{x}](\{\ell_1 = e_1, \ldots, \ell_n = e_n\} \cong \{\ell_1 = e'_1, \ldots, \ell_n = e'_n\})$.

  By conjunction and that all the Γ'_i are disjoint, we have that for all $i \in [n]$, $\overset{\mathsf{val}}{\forall}_{\Gamma'_i}.\sigma_i$. Then by the inductive hypotheses we have that $[\vec{v}/\vec{x}](e_i \cong e'_i : \tau_i)$. Let $[\vec{v}/\vec{x}]e_i \Downarrow v_i$ and $[\vec{v}/\vec{x}]e'_i \Downarrow v'_i$. By Lemma 5 we have that for all p_i where $p_i :: \tau_i \dashv \Gamma_i$, either $v_i \mathrel{/\!/} p_i \dashv b_i$ and $v'_i \mathrel{/\!/} p_i \dashv b_i$ or $v'_i \not\mathrel{/\!/} p_i b_i$.

  To appeal to EQ$_2$, let p be an arbitrary pattern such that $p :: \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \dashv \Gamma'$. We proceed by cases:

- In the case that for all $i \in [n]$ $v_i \,/\!\!/\, p_i \dashv b$ and $v'_i \,/\!\!/\, p_i \dashv b$, by Match$_3$ we have $\{\ell_1 = v_1, \ldots, \ell_n = v_n\} \,/\!\!/\, p \dashv b$ and $\{\ell_1 = v'_1, \ldots, \ell_n = v'_n\} \,/\!\!/\, p \dashv b$.
- In the case that there is some $i \in [n]$ where $v_i \not/\!\!\!/\, p_i$ and $v'_i \not/\!\!\!/\, p_i$, by Match$_4$ we have $\{\ell_1 = v_1, \ldots, \ell_n = v_n\} \not/\!\!\!/\, p$ and $\{\ell_1 = v'_1, \ldots, \ell_n = v'_n\} \not/\!\!\!/\, p$.

Since in all cases either $\{\ell_1 = v_1, \ldots, \ell_n = v_n\} \,/\!\!/\, p \dashv b$ and $\{\ell_1 = v'_1, \ldots, \ell_n = v'_n\} \,/\!\!/\, p \dashv b$ or $\{\ell_1 = v_1, \ldots, \ell_n = v_n\} \not/\!\!\!/\, p$ and $\{\ell_1 = v'_1, \ldots, \ell_n = v'_n\} \not/\!\!\!/\, p$, by EQ$_2$, we may conclude

$$[\vec{v}/\vec{x}](\{\ell_1 = e_1, \ldots, \ell_n = e_n\} \cong \{\ell_1 = e'_1, \ldots, \ell_n = e'_n\})$$

- Iso$_3$: Let $\Gamma = \vec{x} : \vec{\tau}$ and let $\vec{v}$ be arbitrary where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$. Assume $[\vec{v}/\vec{x}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right)$.

  It must be shown that $[\vec{v}/\vec{x}](e_1 \cdot \ell_i \cong e_2 \cdot \ell_i)$.

  By the inductive hypothesis we have $\overset{\mathsf{val}}{\forall}_{\Gamma}.\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } e_1 \cong e_2\right)$, and therefore

$$[\vec{v}/\vec{x}]\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } e_1 \cong e_2\right)$$

  equivalently,

$$\text{if } [\vec{v}/\vec{x}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } [\vec{v}/\vec{x}](e_1 \cong e_2)$$

  As we have $[\vec{v}/\vec{x}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right)$ by assumption, we may conclude

$$[\vec{v}/\vec{x}](e_1 \cong e_2)$$

  As extensional equivalence is the same as contextual equivalence, from this we may conclude

$$[\vec{v}/\vec{x}](e_1 \cdot \ell_i \cong e_2 \cdot \ell_i)$$

- Iso$_4$: For the same reasons as in the proof for Iso$_3$, we have that $[\vec{v}/\vec{x}](e_1 \cong e_2)$. As extensional equivalence is the same as contextual equivalence, we may then conclude

$$[\vec{v}/\vec{x}](\ell_{i,j} \cdot e_1 \cong \ell_{i,j} \cdot e_2)$$

- Iso$_5$: Let $\Gamma = \vec{x} : \vec{\tau}$ and let $\vec{v}$ be arbitrary where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$. Assume $[\vec{v}/\vec{x}]\overset{\mathsf{val}}{\forall}_{\Gamma_1,\Gamma'_1,\ldots,\Gamma_n,\Gamma'_n}.\ \wedge_{i\in[n]}\,(e = p'_i \Rightarrow \sigma_i)$.

  It must be shown that $[\vec{v}/\vec{x}]\,(\texttt{case}\ e\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\}) \cong [\vec{v}/\vec{x}]e'$.

  By inversion of the statics we have that $\Gamma \vdash e : \tau'$; by assumption we have that e atomic and e is in weak head normal form. Therefore by Lemma 7 we have that $[\vec{v}/\vec{x}]e$ val.

  Since case expressions are enforced to be exhaustive, there must be some p_i such that $[\vec{v}/\vec{x}]e \,/\!\!/\, p_i \dashv b$. By Lemma 3 let $[\vec{v}/\vec{x}]e \,/\!\!/\, p'_i \dashv b'$.

By our assumption and the semantics of conjunction we have that

$$[\vec{v}/\vec{x}]\overset{\mathsf{val}}{\forall}_{\Gamma_1,\Gamma_1',\ldots,\Gamma_n,\Gamma_n'}.\,(e = p_i' \Rightarrow \sigma_i)$$

Since e only contains variables in Γ, p_i' only contains variables in Γ_i, and σ_i only contains variables in Γ Γ_i and Γ_i', this can be strengthened to:

$$[\vec{v}/\vec{x}]\overset{\mathsf{val}}{\forall}_{\Gamma_i,\Gamma_i'}.\,(e = p_i' \Rightarrow \sigma_i)$$

Since Γ, Γ_i, and Γ_i' are disjoint, this is equivalent to:

$$\overset{\mathsf{val}}{\forall}_{\Gamma_i,\Gamma_i'}.\,([\vec{v}/\vec{x}]e = [\vec{v}/\vec{x}]p_i' \Rightarrow [\vec{v}/\vec{x}]\sigma_i)$$

Then since p_i only contains variables in Γ_i (and therefore none of $\vec{x}$), we have: Since Γ, Γ_i, and Γ_i' are disjoint, this is equivalent to:

$$\overset{\mathsf{val}}{\forall}_{\Gamma_i,\Gamma_i'}.\,([\vec{v}/\vec{x}]e = p_i' \Rightarrow [\vec{v}/\vec{x}]\sigma_i)$$

Since p_i' contains only variables in Γ_i, we may partially invoke this result with b' which gives us:

$$[b']\overset{\mathsf{val}}{\forall}_{\Gamma_i'}.\,([\vec{v}/\vec{x}]e = p_i' \Rightarrow [\vec{v}/\vec{x}]\sigma_i)$$

Since Γ_i and Γ_i' are disjoint and since e contains no variables in Γ_i, we can rearrange this to get:

$$\overset{\mathsf{val}}{\forall}_{\Gamma_i'}.\,([\vec{v}/\vec{x}]e = [b']p_i' \Rightarrow [b'][\vec{v}/\vec{x}]\sigma_i)$$

By Lemma 4, $[\vec{v}/\vec{x}]e = [b']p_i'$ is true so this is equivalent to:

$$\overset{\mathsf{val}}{\forall}_{\Gamma_i'}.\,([b'][\vec{v}/\vec{x}]\sigma_i)$$

Once again rearranging because Γ, Γ_i, and Γ_i' are disjoint, we get: $[\vec{v}/\vec{x}][b']\overset{\mathsf{val}}{\forall}_{\Gamma_i'}.\sigma_i$.

This allows us to invoke the inductive hypothesis to get: $[\vec{v}/\vec{x}][b']e_i' \cong [\vec{v}/\vec{x}][b']e' : \tau$.

Since the variables in b' don't appear in e', this is equivalent to: $[\vec{v}/\vec{x}][b']e_i' \cong [\vec{v}/\vec{x}]e' : \tau$.

By Lemma 3, this is equivalent to: $[\vec{v}/\vec{x}][b]e_i \cong [\vec{v}/\vec{x}]e' : \tau$.

By $\textsc{Dyn}_9$, $[\vec{v}/\vec{x}]\,(\texttt{case}\ e\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\}) \Longrightarrow [\vec{v}/\vec{x}][b]e_i$.

Therefore since extensional equivalence is closed under evaluation, $[\vec{v}/\vec{x}]\,(\texttt{case}\ e\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\})$ $[\vec{v}/\vec{x}]e' : \tau$.

- $\textsc{Iso}_6$: By symmetry and $\textsc{Iso}_5$.

- $\textsc{Iso}_7$ Let $\Gamma = \vec{y} : \vec{\tau}$ and let $\vec{v}$ be arbitrary where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$. Assume $[\vec{v}/\vec{y}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma',x:\tau',\Gamma''}.\sigma \wedge \sigma'\right)$.

It must be shown that $[\vec{v}/\vec{y}](\texttt{case}\ e\ \{\ldots\} \cong \texttt{case}\ e'\ \{\ldots\})$.

By the inductive hypothesis we have

$$\overset{\mathsf{val}}{\forall}_{\Gamma}.\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } e \cong e'\right)$$

and therefore

$$\left(\text{if } [\vec{v}/\vec{y}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } [\vec{v}/\vec{y}](e \cong e')\right)$$

Since $[\vec{v}/\vec{y}]\sigma$ only contains variables in Γ', by assumption and conjunction we already have $[\vec{v}/\vec{y}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right)$. Therefore we may conclude $[\vec{v}/\vec{y}](e \cong e')$.

Let $e \Downarrow w$. By the inductive hypothesis we have

$$\overset{\mathsf{val}}{\forall}_{\Gamma,x:\tau'}.\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma''}.\sigma'\right) \text{ then } \texttt{case}\ x\ \{\ldots\} \cong \texttt{case}\ x\ \{\ldots'\}\right)$$

and therefore since x is fresh,

$$\overset{\mathsf{val}}{\forall}_{x:\tau'}.\left(\text{if } [\vec{v}/\vec{y}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma''}.\sigma'\right) \text{ then } [\vec{v}/\vec{y}](\texttt{case}\ x\ \{\ldots\} \cong \texttt{case}\ x\ \{\ldots'\})\right)$$

Invoking this with w, we have

$$\text{if } [w/x][\vec{v}/\vec{y}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma''}.\sigma'\right) \text{ then } [\vec{v}/\vec{y}](\texttt{case}\ w\ \{\ldots\} \cong \texttt{case}\ w\ \{\ldots'\})$$

By assumption and conjunction we already have $[w/x][\vec{v}/\vec{y}]\left(\overset{\mathsf{val}}{\forall}_{\Gamma''}.\sigma'\right)$. Therefore we may conclude

$$[\vec{v}/\vec{y}](\texttt{case}\ w\ \{\ldots\} \cong \texttt{case}\ w\ \{\ldots'\})$$

As $e \cong w$ and therefore $e' \cong w'$ by transitivity, by referential transparency we have Therefore we may conclude

$$[\vec{v}/\vec{y}](\texttt{case}\ e\ \{\ldots\} \cong \texttt{case}\ e'\ \{\ldots'\})$$

- $\text{I}\textsc{so}_8$: Let $\Gamma = \vec{y} : \vec{\tau}$ and let $\vec{v}$ be arbitrary where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$. Assume $[\vec{v}/\vec{y}]\left(\overset{\mathsf{val}}{\forall}_{x:\tau,\Gamma'}.\sigma\right)$.

  It must be shown that $[\vec{v}/\vec{x}](\lambda x_1.e_1 \cong \lambda x_2.e_2)$

  To appeal to EQ_1, take arbitrary w such that $\Gamma \vdash w : \tau$ and w val. By the inductive hypothesis we have

$$\overset{\mathsf{val}}{\forall}_{\Gamma,x:\tau}.\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } e_1 \cong e_2 : \tau'\right)$$

and therefore since x is fresh,

$$\text{if } [w/x][\vec{v}/\vec{y}] \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } [w/x][\vec{v}/\vec{y}]e_1 \cong e_2 : \tau'$$

By assumption, we already have $[w/x][\vec{v}/\vec{y}] \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right)$. Therefore we have

$$[w/x][\vec{v}/\vec{y}]e_1 \cong e_2 : \tau'$$

By $\textsc{Dyn}_{13}$, we have

$$[\vec{v}/\vec{y}](\lambda x_1.e_1)\ w \Longrightarrow [\vec{v}/\vec{y}][w/x_1]e_1 = [\vec{v}/\vec{y}][w/x][x/x_1]e_1$$

and

$$[\vec{v}/\vec{y}](\lambda x_2)\ w \Longrightarrow [\vec{v}/\vec{y}][w/x_2]e_2 = [\vec{v}/\vec{y}][w/x][x/x_2]e_2$$

. Therefore since $[w/x][\vec{v}/\vec{y}]e_1 \cong e_2 : \tau'$ and x is fresh, we have that

$$[\vec{v}/\vec{y}](\lambda x_1.e_1 \cong \lambda x_2.e_2)$$

- $\textsc{Iso}_9$: Let $\Gamma = \vec{x} : \vec{\tau}$ and let $\vec{v}$ be arbitrary where $v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$. Assume $[\vec{v}/\vec{x}] \left(\overset{\mathsf{val}}{\forall}_{\Gamma',\Gamma''}.\sigma \wedge \sigma'\right)$.

  It must be shown that $[\vec{v}/\vec{x}](e_1\ e_1' \cong e_2\ e_2')$.

  By the inductive hypothesis we have

  $$\overset{\mathsf{val}}{\forall}_{\Gamma}.\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } e_1 \cong e_2 : \tau \to \tau'\right)$$

  and therefore

  $$\text{if } [\vec{v}/\vec{x}] \left(\overset{\mathsf{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } [\vec{v}/\vec{x}](e_1 \cong e_2 : \tau \to \tau')$$

  As σ does not contain any variables in Γ'', by assumption and conjunction we already have $[\vec{v}/\vec{x}] \left(\overset{\mathsf{val}'}{\forall}_{\Gamma}.\sigma\right)$ therefore we may conclude

  $$[\vec{v}/\vec{x}](e_1 \cong e_2 : \tau \to \tau')$$

  Similarly, by the inductive hypothesis we have

  $$\overset{\mathsf{val}}{\forall}_{\Gamma}.\left(\text{if } \left(\overset{\mathsf{val}}{\forall}_{\Gamma''}.\sigma'\right) \text{ then } e_1' \cong e_2' : \tau\right)$$

  and therefore

  $$\text{if } [\vec{v}/\vec{x}] \left(\overset{\mathsf{val}}{\forall}_{\Gamma''}.\sigma'\right) \text{ then } [\vec{v}/\vec{x}](e_1' \cong e_2' : \tau)$$

As σ' does not contain any variables in Γ', by assumption and conjunction we already have $[\vec{v}/\vec{x}]\left(\overset{\mathsf{val}''}{\forall}_{\Gamma}.\sigma'\right)$ therefore we may conclude

$$[\vec{v}/\vec{x}](e_1' \cong e_2' : \tau)$$

Since $[\vec{v}/\vec{x}]e_1 \cong [\vec{v}/\vec{x}]e_2 : \tau \to \tau'$, by inversion of EQ_1 we have

$$[\vec{v}/\vec{x}](e_1\ e_1') \cong [\vec{v}/\vec{x}](e_2\ e_1') : \tau'$$

Since $[\vec{v}/\vec{x}](e_1' \cong e_2' : \tau)$, by referential transparency we then have

$$[\vec{v}/\vec{x}](e_1\ e_1') \cong [\vec{v}/\vec{x}](e_2\ e_2') : \tau'$$

- $\textsc{Iso}_{10}$: As fixed points may be expressed solely in terms of lambdas (as demonstrated by the Y combinator), this is a special case of $\textsc{Iso}_8$.

Chapter 7

Implementation

We've implemented Zeus as a grading assistant to group Standard ML programs into equivalence classes. The implementation is written in Standard ML. Student submissions often contain declarations of many different expressions, as they are often asked to implement their solutions to many different homework problems in the same code file. Rather than checking equivalence of entire files, it is more effective to check equivalence of student files on a per-problem basis. In practice, each homework problem consists of a function which students are asked to implement. A particular problem can then be identified by the name of the function it asks for implementation. The grading assistant is therefore invoked with two arguments: a path to a directory in which all of the student submissions are contained, and the name of the function we wish to detect equivalence of. Each invocation of the algorithm produces output which helps grade a particular problem, so to grade an entire assignment the algorithm is invoked once per problem on the assignment.

The implementation is written to group Standard ML programs into equivalence classes. It does this by transpiling each code submission from Standard ML into LambdaPix, and then running the Zeus algorithm pairwise.

7.1 Results

We present the results of using Zeus to group submissions to a homework assignment in an introductory functional programming course taught in Standard ML.

Figure 7.1 shows the degree to which Zeus reduces grading. Starting with 186 student submissions, running Zeus on various problems resulted in around 52.8 equivalence classes on average, which corresponds to roughly a 72% reduction on average in the amount of grading work there is to be done.

Figure 7.2 shows the sizes of the six largest equivalence classes for each of the problems. On average around 105 submissions get put into the largest equivalence class, which is roughly 56% of them. After that, the class sizes taper off in a pattern resembling exponential decay.

Function	**Equivalence Classes**	**Reduction in Grading Time**
heads	22	88%
tails	54	71%
lookAndSay	92	51%
prefixSum	34	82%
prefixSumFast	62	67%

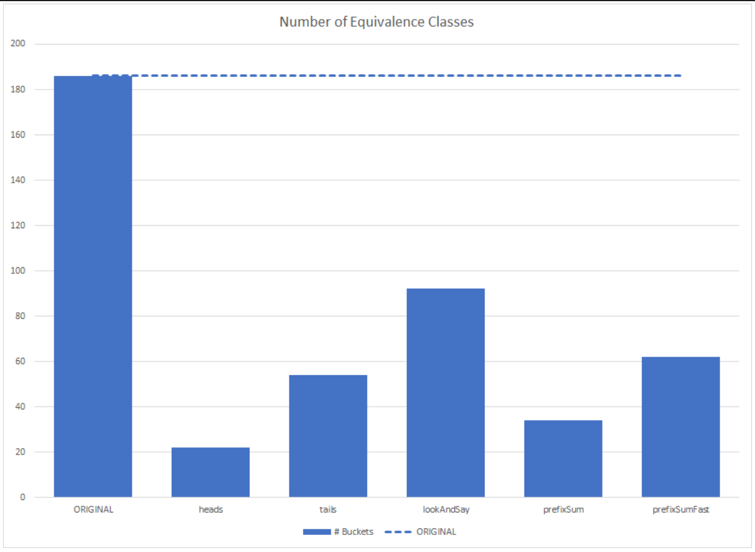

Figure 7.1: Number of equivalence classes identified by Zeus (originally 186 items)

Function	**Class 1**	**Class 2**	**Class 3**	**Class 4**	**Class 5**	**Class 6**
heads	150	9	4	3	2	1
tails	112	9	8	3	2	2
lookAndSay	48	22	6	6	4	4
prefixSum	134	6	6	4	3	2
prefixSumFast	82	13	9	8	6	6

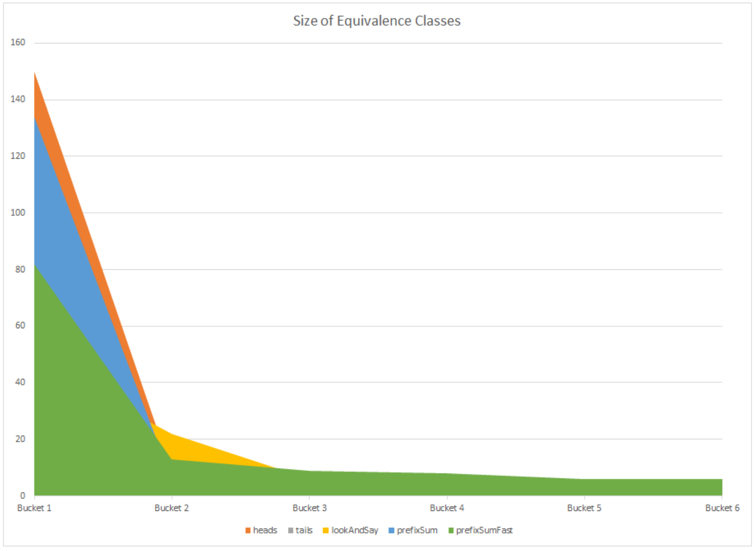

Figure 7.2: Sizes of largest equivalence classes (originally 186 items)

References

[1] Umut A. Acar, Amal Ahmed, and Matthias Blume. “Imperative Self-Adjusting Computation”. In: *POPL* (2008).

[2] Karl Crary. “Logical Relations and a Case Study in Equivalence Checking”. In: *Advanced Topics in Types and Programming Languages.* Ed. by Benjamin C. Pierce. The MIT Press, 2005. Chap. 6, pp. 223–244.

[3] Benny Godlin and Ofer Strichman. “Inference Rules for Proving the Equivalence of Recursive Functions”. In: *Acta Informatica* (2010).

[4] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. “Automated Clustering and Program Repair for Introductory Programming Assignments”. In: *PLDI* (2018).

[5] Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, 2016.

[6] David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong. “Hector: An Equivalence Checker for a Higher-Order Fragment of ML”. In: *International Conference on Computer Aided Verification* (2012).

[7] Nuno P. Lopes and Jose Monteiro. “Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic”. In: *International Journal on Software Tools for Technology Transfer* (2016).

[8] Georce C. Necula. “Translation Validation for an Optimizing Compiler”. In: *ACM SIGPLAN Notices* (2000).

[9] Christopher A. Stone and Robert Harper. “Extensional Equivalence and Singleton Types”. In: *ACM Transactions on Computational Logic* (2006).

[10] Sorin Lerner Sudipta Kundu Zachary Tatlock. “Proving Optimizations Correct using Paramterized Program Equivalence”. In: *PLDI* (2009).