# Program Equivalence for Assisted Grading of Functional Programs

JOSHUA CLUNE, Carnegie Mellon University
VIJAY RAMAMURTHY, Carnegie Mellon University
RUBEN MARTINS, Carnegie Mellon University
UMUT A. ACAR, Carnegie Mellon University

In courses that involve programming assignments, giving meaningful feedback to students is an important challenge. Human beings can give useful feedback by manually grading the programs but this is a time-consuming, labor intensive, and usually boring process. Automatic graders can be fast and scale well but they usually provide poor feedback. Although there has been research on improving automatic graders, research on scaling and improving human grading is limited.

We propose to scale human grading by augmenting the manual grading process with an equivalence algorithm that can identify the equivalences between student submissions. This enables human graders to give targeted feedback for multiple student submissions at once. Our technique is conservative in two aspects. First, it identifies equivalence between submissions that are algorithmically similar, e.g., it cannot identify the equivalence between quicksort and mergesort. Second, it uses formal methods instead of clustering algorithms from the machine learning literature. This allows us to prove a soundness result that guarantees that submissions will never be clustered together in error. Despite only reporting equivalence when there is algorithmic similarity and the ability to formally prove equivalence, we show that our technique can significantly reduce grading time for thousands of programming submissions from an introductory functional programming course.

Additional Key Words and Phrases: Program Equivalence, Assisted Grading, Formal Methods, Functional Programming

## 1 INTRODUCTION

There have been many efforts to develop techniques for automated reasoning of programming assignments at scale. This has lead to the rise of automatic graders, programs that take in a set of student submissions and output grades or feedback for those submissions without requiring any human input. While recent years have yielded substantial improvements in automatic grading techniques [Gulwani et al. 2018; Kaleeswaran et al. 2016; Liu et al. 2019; Perry et al. 2019; Singh et al. 2013; Wang et al. 2018], automatic graders are still more limited in the feedback they can provide than human graders.

This creates a trade-off between scale and quality. For small courses, it makes sense to utilize human graders in order to provide the best feedback possible. For Massive Open Online Courses, human involvement in grading all submissions is often logistically impossible, so it makes sense to use automatic graders. But neither option is ideal for large, in-person, introductory functional courses. When introductory functional courses use automatic graders, it hurts the students because they receive less targeted feedback, and it can hurt the teaching staff to lose a valuable avenue for addressing uncommon misunderstandings. But when introductory functional courses use human graders, it creates a large burden on the teaching staff, and it may require capping the size of the class, hurting students by limiting their opportunity to take the class.

Authors' addresses: Joshua Clune, Carnegie Mellon University, josh.seth.clune@gmail.com; Vijay Ramamurthy, Carnegie Mellon University, vrama628@gmail.com; Ruben Martins, Carnegie Mellon University, rubenm@andrew.cmu.edu; Umut A. Acar, Carnegie Mellon University, umut@cs.cmu.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

To provide an option that eases the cost of human grading without sacrificing feedback quality, we propose a method of enabling human graders to give targeted feedback to multiple students at once. Our approach takes a pair of expressions submitted by students and deconstructs them simultaneously to build up a formula that is valid only if the expressions are equivalent. This pairwise equivalence test is used to cluster student submissions into buckets for which all submissions can be graded and given feedback simultaneously. Our approach recognizes expressions as equivalent by finding equivalences in each expression's subexpressions. To do this, it uses a variety of inference rules to simultaneously deconstruct the expressions down to their atomic subexpressions. It then outputs formulas that are valid only if the atomic subexpressions are equivalent. Finally, our inference rules recursively use the formulas of these subexpressions as subformulas to build up a larger formula that indicates the equivalence of the overall expression. This final formula's validity can be checked by an SMT Solver to determine whether the two expressions are equivalent.

A central benefit of our approach is that when two expressions are recognized as equivalent, this fact does not merely reflect that the two expressions produce the same outputs on shared inputs. In input/output grading, the correctness of code is determined entirely by whether a student submission produces correct outputs when given a large and diverse set of inputs. But in our approach, all equivalences arise from similarities in subexpressions, so equivalences found by our technique are discoverable only due to underlying algorithmic similarities. This enables instructors to give feedback based not only on whether a problem was solved correctly, but based on the algorithmic decisions that were involved in the student's solution.

Three primary factors that impact the grading and feedback of student programs are correctness, algorithmic approach, and style. While our approach is meant to enable providing better feedback concerning algorithmic approach, as opposed to simply providing feedback concerning correctness as in input/output grading, evaluating style is outside of the scope of our technique. For that reason, we believe that our approach is best utilized in conjunction with the methods courses already use to evaluate style. For courses already doing automatic grading, this should not be an issue because if they are already doing automatic grading, they are already automatically doing style checking, and can, therefore, use that in conjunction with our approach to provide all of the same style feedback the course already provided, but additionally provide human feedback for algorithmic content.

For courses already doing fully human grading, even if it is still necessary to grade each assignment individually to address style concerns, we believe our approach can make it possible to better allocate human resources for the grading process. A grader focusing entirely on one or two large buckets can be more efficient by not being forced to figure out which common approach is being taken by every individual submission. This can help the grader more quickly move on from understanding the student's solution to addressing any style concerns, and it also helps ensure fairer grading in guaranteeing that the same grader will grade all similar submissions. A grader focusing entirely on grading submissions that were clustered with few if any other programs can anticipate ahead of time that their grading will likely require providing more frequent and/or detailed comments. This can enable course staffs to give more submissions to graders of large buckets, easing the burden of singleton/small bucket graders.

The differences between our approach and other state-of-the-art automatic graders and clustering techniques [Gulwani et al. 2018; Perry et al. 2019; Wang et al. 2018] stem from differences in motivation. Since each bucket generated by our approach is meant to be graded by a human, it is more important for our technique to distinguish nonequivalent submissions than to ensure that all equivalent submissions are placed in the same bucket. Ensuring that all equivalent submissions are placed in the same bucket reduces time spent grading equivalent programs, enabling instructors to spend more time giving detailed feedback. This is an important goal, but it is of lower priority than preserving the accuracy of human feedback because it does not matter how detailed feedback is if it does not apply to the student to whom it is given. To secure the accuracy of human feedback while using our approach, we guarantee the correctness of our technique's recognized equivalences by proving a soundness theorem that states that if our technique recognizes two expressions as equivalent, they necessarily exhibit identical behavior.

In summary, the contributions of our paper are as follows:

- We define an effective and efficient technique for identifying equivalences between purely functional programs. The technique's design ensures that only algorithmically similar programs will be recognized as equivalent.
- We prove the soundness of this technique, showing that if our approach identifies an equivalence between two expressions, it is necessarily the case that the two expressions exhibit identical behavior.
- We implement our approach in a tool called ZEUS and demonstrate its effectiveness in assisting the grading of more than 4,000 student submissions from a functional programming course taught at the college level in Standard ML.

## 2 MOTIVATING EXAMPLES

Our approach is meant to cluster expressions that are algorithmically similar, but potentially syntactically different. In this section, we show two examples of similar implementations of the same function that are successfully identified by our tool as equivalent, and describe one example in which two solutions to a task are not recognized as equivalent due to algorithmic dissimilarities.

```
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)
  ))
```

Fig. 1. Two implementations of adding two optional numbers

Figure 1 contains two functions that take in two `int options` as input, and adds the `ints` in the `options` if possible, returning `NONE` otherwise. The right expression's conditional logic is modeled after Haskell-style monads, interacting with the higher order bind function to case on `x` first, and then potentially `y` depending on the value of `x`, whereas the left expression cases on `x` and `y` simultaneously. Still, our approach is able to fully encode both expressions' conditional logic structures and produce a valid formula. A demonstration of how our approach specifically encodes these conditional logic structures is included in Section 5.

Figure 2 contains two functions that implement mergesort. The left implementation uses a style that emphasizes pattern matching on input arguments while the right implementation uses a style that emphasizes nesting binding structures. Although there are several syntactic differences between the two expressions, both implement the same underlying algorithm. Therefore, our approach recognizes these two functions as equivalent.

```
fun split [] = ([], [])
  | split [x] = ([x], [])
  | split (x::y::L) =
      let
        val (A, B) = split L
      in
        (x::A, y::B)
      end

fun merge([], L) = L
  | merge(L, []) = L
  | merge(x::xs, y::ys) =
      if x < y
      then x :: merge (xs, y::ys)
      else y :: merge (x::xs, ys)

fun msort [] = []
  | msort [x] = [x]
  | msort L =
      let
        val (A, B) = split L
      in
        merge(msort A, msort B)
      end
```

```
fun split [] = ([], [])
  | split (x::xs) =
      case xs of
          [] => ([x], [])
        | (y::ys) =>
            let
              val (A, B) = split ys
            in
              (x::A, y::B)
            end

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)

fun msort [] = []
  | msort [x] = [x]
  | msort L =
      let
        val (A, B) = split L
      in
        merge(msort A, msort B)
      end
```

Fig. 2. Two implementations of mergesort

Our approach is not intended to cluster programs just by correctness, or final input/output behavior, but by structure. This enables our approach to distinguish between correct submissions that use different algorithms. For instance, one of the benchmarks we use in Section 7 to evaluate our tool is a task called slowDoop. The goal of this task is to take in an arbitrary list $L$ and return a list in which all elements in $L$ appear exactly once. Consider a similar task in which the goal is the same but has the added stipulation that the final list must be sorted. A reasonable $O(n^2)$ solution to this task would be to iterate over $L$, only keeping elements that do not appear later in the list, and then sort the result. But a better $O(n \log n)$ solution would be to first sort $L$, and then iterate over the resulting list once to remove duplicate elements. While correct implementations of these algorithms are identical from an input/output perspective, our approach would cluster them separately, and we believe that they merit different feedback.

## 3 LAMBDAPIX

Our approach operates over a language which we call LambdaPix. LambdaPix is designed to be a target for transpilation from functional programming languages such as Standard ML, OCaml, or Haskell. Our techniques apply to purely functional programs only and do not allow for state (e.g.,

| | | | | |
|---|---|---|---|---|
| *base types* | $b$ | $::=$ | $int \mid boolean$ | |
| *types* | $\tau$ | $::=$ | $b$ | *base type* |
| | | $\mid$ | $\delta$ | *data type* |
| | | $\mid$ | $\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$ | *product type* |
| | | $\mid$ | $\tau_1 \rightarrow \tau_2$ | *function type* |
| *injection labels* | $i$ | $::=$ | $\mathrm{label}_1 \mid \mathrm{label}_2 \mid \ldots$ | |
| *patterns* | $p$ | $::=$ | $\_$ | *wildcard pattern* |
| | | $\mid$ | $x$ | *variable pattern* |
| | | $\mid$ | $\{\ell_1 = p_1, \ldots, \ell_n = p_n\}$ | *record pattern* |
| | | $\mid$ | $x \ \mathtt{as}\ p$ | *alias pattern* |
| | | $\mid$ | $c$ | *constant pattern* |
| | | $\mid$ | $i \cdot p$ | *injection pattern (with argument)* |
| | | $\mid$ | $i$ | *injection pattern (without argument)* |
| *primitive operations* | $o$ | $::=$ | $+ \mid - \mid * \mid < \mid > \mid \leq \mid \geq$ | |
| *expressions* | $e$ | $::=$ | $c$ | *constant* |
| | | $\mid$ | $x$ | *variable* |
| | | $\mid$ | $\{\ell_1 = e_1, \ldots, \ell_n = e_n\}$ | *record* |
| | | $\mid$ | $e \cdot \ell_i$ | *projection* |
| | | $\mid$ | $i \cdot e$ | *injection (with argument)* |
| | | $\mid$ | $i$ | *injection (without argument)* |
| | | $\mid$ | $\mathtt{case}\ e\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\}$ | *case analysis* |
| | | $\mid$ | $\lambda x.e$ | *abstraction* |
| | | $\mid$ | $e_1\ e_2$ | *application* |
| | | $\mid$ | $\mathtt{fix}\ x\ \mathtt{is}\ e$ | *fixed point* |
| | | $\mid$ | $o$ | *primitive operation* |

Fig. 3. The syntax of LambdaPix

references) but are otherwise unrestricted and make no further assumptions about the programs. In this section, we present the syntax and semantics for LambdaPix.

LambdaPix is so named because it is the lambda calculus enriched with pattern matching and fixed 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.

We give the syntax for LambdaPix in Figure 3. We use meta-variables $x$, $y$, and $z$ (and variants) to range over an unspecified set of variables and use the meta-variable $i$ to range over a separate, unspecified list of injection labels.

### 3.1 Static Semantics

We assume an arbitrary fixed set of disjoint algebraic data types with unique associated injection labels (by unique, it is meant that there are no shared injection labels between distinct data types). In particular, we assume a fixed set of judgments of the form $i : \tau \hookrightarrow \delta$ for injection labels that take in an argument of type $\tau$ to produce an expression of data type $\delta$, and a fixed set of judgments of the form $i : \delta$ for injection labels of data type $\delta$ that do not take in an argument. We take $i : \tau \hookrightarrow \delta$ to mean that the type $\delta$ has a label $i$ which accepts an argument of type $\tau$, and we take $i : \delta$ to mean that the type $\delta$ has a label $i$ that does not accept an argument. Note that by allowing $\tau$ to contain instances of $\delta$, this data type system affords LamdbaPix a form of inductive types.

Figure 4 defines an auxiliary judgment used in the typechecking of expressions: pattern typing. The pattern typing judgment $p :: \tau \dashv \Gamma$ defines that expressions of type $\tau$ can be matched against

$$\frac{}{\_ :: \tau \dashv}\ \textsc{PatTy}_1 \qquad \frac{}{x :: \tau \dashv x : \tau}\ \textsc{PatTy}_2 \qquad \frac{p_1 :: \tau_1 \dashv \Gamma_1 \quad \dots \quad p_n :: \tau_n \dashv \Gamma_n}{\{l_1 = p_1, \dots, l_n = p_n\} :: \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\} \dashv \Gamma_1 \dots \Gamma_n}\ \textsc{PatTy}_3$$

$$\frac{p :: \tau \dashv \Gamma}{x \;\mathsf{as}\; p :: \tau \dashv \Gamma, x : \tau}\ \textsc{PatTy}_4 \qquad \frac{}{c :: b \dashv}\ \textsc{PatTy}_5 \qquad \frac{i : \delta}{i :: \delta \dashv}\ \textsc{PatTy}_6 \qquad \frac{i : \tau \hookrightarrow \delta \quad p :: \tau \dashv \Gamma}{i \cdot p :: \delta \dashv \Gamma}\ \textsc{PatTy}_7$$

Fig. 4. Pattern typing in LambdaPix

the pattern $p$, and that doing so produces new variable bindings whose types are captured in $\Gamma$. This is used in the typechecking of case expressions.

$$\frac{}{\Gamma \vdash c : b}\ \textsc{Ty}_1 \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Ty}_2 \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\ell_1 = e_1, \dots, \ell_n = e_n\} : \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\}}\ \textsc{Ty}_3$$

$$\frac{\Gamma \vdash e : \{\dots, \ell_i : \tau_i, \dots\}}{\Gamma \vdash e \cdot \ell_i : \tau_i}\ \textsc{Ty}_4 \qquad \frac{i : \delta}{\Gamma \vdash i : \delta}\ \textsc{Ty}_5 \qquad \frac{i : \tau \hookrightarrow \delta \quad \Gamma \vdash e : \tau}{\Gamma \vdash i \cdot e : \delta}\ \textsc{Ty}_6$$

$$\frac{\Gamma \vdash e : \tau \quad p_1 :: \tau \dashv \Gamma_1 \quad \Gamma, \Gamma_1 \vdash e_1 : \tau' \quad \dots \quad p_n :: \tau \dashv \Gamma_n \quad \Gamma, \Gamma_n \vdash e_n : \tau'}{\Gamma \vdash \mathsf{case}\; e\; \{p_1.e_1 \mid \dots \mid p_n.e_n\} : \tau'}\ \textsc{Ty}_7 \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \textsc{Ty}_8$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\; e_2 : \tau_2}\ \textsc{Ty}_9 \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\; x\; \mathsf{is}\; e : \tau}\ \textsc{Ty}_{10} \qquad \frac{}{\Gamma, o : \tau_1 \to \tau_2 \vdash o : \tau_1 \to \tau_2}\ \textsc{Ty}_{11}$$

Fig. 5. Expression typing in LambdaPix

Figure 5 defines typing for expressions in LambdaPix.

*Definition 3.1 (Well-formed).* A LambdaPix expression $e$ is well-formed if there exists a type $\tau$ such that $\Gamma_{\text{initial}} \vdash e : \tau$ is derivable from Figure 5's typing rules, where $\Gamma_{\text{initial}}$ only contains typing judgments for primitive operations of the form $o : \tau_1 \to \tau_2$.

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.

## 3.2 Dynamic Semantics

Here we define how LambdaPix expressions evaluate. We define evaluation as a small-step dynamic semantics where the judgment $e \mapsto e'$ means that $e$ steps to $e'$ and the judgment $e$ val means that $e$ is a value and doesn't step any further. LambdaPix enjoys progress and preservation.

*Definition 3.2 (Progress and Preservation).* For any typing context $\Gamma$ and expression $e$ such that $\Gamma \vdash e : \tau$ it is either the case that there exists an expression $e'$ such that $e \mapsto e'$ (in which case $\Gamma \vdash e' : \tau$ as well) or $e$ val.

LambdaPix also enjoys the finality of values: it is never the case that both $e \mapsto e'$ and $e$ val.

To define evaluation we first define two helper judgments to deal with pattern matching (Figure 6). The judgment $v \mathbin{/\!\!/} p \dashv B$ means the value $v$ matches to the pattern $p$ producing $B$, where $B$ is a set of bindings of the form $v'/x$ that indicate the value $v'$ is bound to the variable $x$. The judgment $v \mathbin{/\!\!/\!\!\!\!\!\setminus}\, p$ means the expression $v$ does not match to the pattern $p$. It is assumed as a precondition to

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

$$\frac{v_1 \mathbin{/\!\!/} p_1 \dashv B_1 \quad \dots \quad v_n \mathbin{/\!\!/} p_n \dashv B_n}{\{\ell_1 = v_1, \dots, \ell_n = v_n\} \mathbin{/\!\!/} \{\ell_1 = p_1, \dots, \ell_n = p_n\} \dashv B_1 \dots B_n}\ \textsc{Match}_3$$

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

$$\frac{v \not\mathbin{/\!\!/} p}{v \not\mathbin{/\!\!/} x \text{ as } p}\ \textsc{Match}_6 \qquad \frac{c_1 = c_2}{c_1 \mathbin{/\!\!/} c_2 \dashv}\ \textsc{Match}_7 \qquad \frac{c_1 \neq c_2}{c_1 \not\mathbin{/\!\!/} c_2}\ \textsc{Match}_8 \qquad \frac{}{i \mathbin{/\!\!/} i \dashv}\ \textsc{Match}_9$$

$$\frac{i_1 \neq i_2}{i_1 \not\mathbin{/\!\!/} i_2}\ \textsc{Match}_{10} \qquad \frac{v \mathbin{/\!\!/} p \dashv B}{i \cdot v \mathbin{/\!\!/} i \cdot p \dashv B}\ \textsc{Match}_{11} \qquad \frac{i_1 \neq i_2}{i_1 \cdot v \not\mathbin{/\!\!/} i_2 \cdot p}\ \textsc{Match}_{12} \qquad \frac{v \not\mathbin{/\!\!/} p}{i \cdot v \not\mathbin{/\!\!/} i \cdot p}\ \textsc{Match}_{13}$$

$$\frac{}{i_1 \cdot v \not\mathbin{/\!\!/} i_2}\ \textsc{Match}_{14} \qquad \frac{}{i_1 \not\mathbin{/\!\!/} i_2 \cdot p}\ \textsc{Match}_{15}$$

Fig. 6. Pattern matching in LambdaPix

these judgements that $v$ val, $\vdash v : \tau$, and $p :: \tau$. Pattern matching in LambdaPix enjoys the property that for any $v$ and $p$ satisfying the above preconditions it is either the case that there exist bindings $B$ such that $v \mathbin{/\!\!/} p \dashv B$, or $v \not\mathbin{/\!\!/} p$. It is never simultaneously the case that $v \mathbin{/\!\!/} p \dashv B$ and $v \not\mathbin{/\!\!/} p$.

$$\frac{}{c \text{ val}}\ \textsc{Dyn}_1 \qquad \frac{e_1 \text{ val} \quad e_2 \text{ val} \quad \dots \quad e_{i-1} \text{ val} \quad e_i \mapsto e_i'}{\{\dots, \ell_i = e_i, \dots\} \mapsto \{\dots, \ell_i = e_i', \dots\}}\ \textsc{Dyn}_2 \qquad \frac{e_1 \text{ val} \quad \dots \quad e_n \text{ val}}{\{\ell_1 = e_1, \dots, \ell_n = e_n\} \text{ val}}\ \textsc{Dyn}_3$$

$$\frac{e \mapsto e'}{e \cdot \ell_i \mapsto e' \cdot \ell_i}\ \textsc{Dyn}_4 \qquad \frac{\{\dots, \ell_i = e_i, \dots\} \text{ val}}{\{\dots, \ell_i = e_i, \dots\} \cdot \ell_i \mapsto e_i}\ \textsc{Dyn}_5 \qquad \frac{e \mapsto e'}{i \cdot e \mapsto i \cdot e'}\ \textsc{Dyn}_6 \qquad \frac{e \text{ val}}{i \cdot e \text{ val}}\ \textsc{Dyn}_7$$

$$\frac{}{i \text{ val}}\ \textsc{Dyn}_8 \qquad \frac{e \mapsto e'}{\text{case } e\ \{p_1.e_1 \mid \dots \mid p_n.e_n\} \mapsto \text{case } e'\ \{p_1.e_1 \mid \dots \mid p_n.e_n\}}\ \textsc{Dyn}_9$$

$$\frac{e \text{ val} \quad e \not\mathbin{/\!\!/} p_1 \quad \dots \quad e \not\mathbin{/\!\!/} p_{i-1} \quad e \mathbin{/\!\!/} p_i \dashv B}{\text{case } e\ \{\dots \mid p_i.e_i \mid \dots\} \mapsto [B]e_i}\ \textsc{Dyn}_{10} \qquad \frac{}{\lambda x.e \text{ val}}\ \textsc{Dyn}_{11} \qquad \frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \textsc{Dyn}_{12}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \textsc{Dyn}_{13} \qquad \frac{e_2 \text{ val}}{(\lambda x.e)\ e_2 \mapsto [e_2/x]e}\ \textsc{Dyn}_{14} \qquad \frac{}{\texttt{fix } x \texttt{ is } e \mapsto [\texttt{fix } x \texttt{ is } e/x]e}\ \textsc{Dyn}_{15}$$

$$\frac{}{o \text{ val}}\ \textsc{Dyn}_{16} \qquad \frac{e \text{ val}}{o\ e \mapsto e'}\ \textsc{Dyn}_{17} \qquad \frac{v \text{ val}}{v \Mapsto v}\ \textsc{BigDyn}_1 \qquad \frac{e \mapsto e' \quad e' \Mapsto v}{e \Mapsto v}\ \textsc{BigDyn}_2$$

Fig. 7. Dynamic semantics of LambdaPix

In Figure 7 we use these helper judgments to define the evaluation judgments. In $\textsc{Dyn}_{17}$, $e'$ is meant to be understood as a hard-coded value dependent on the primitive operation $o$. We use these judgments to define what it means for an expression to evaluate to a value. We use $e \Mapsto v$ to denote that expression $e$ evaluates to value $v$. In rules $\textsc{BigDyn}_1$ and $\textsc{BigDyn}_2$, big-step dynamics are defined as the transitive closure of the small-step dynamics.

## 4 SOUND EQUIVALENCE INFERENCES

Our approach takes as input two LambdaPix expressions of the same type and outputs a logic formula which is valid only if the two expressions are equivalent. We construct this logic formula by constructing a proof tree of sound equivalence inferences.

### 4.1 Logic Formulas

$$\begin{array}{rcll} \sigma & ::= & t_1 \equiv t_2 & \textit{term equivalence} \\ & & \sigma_1 \wedge \sigma_2 & \textit{conjunction} \\ & & \sigma_1 \vee \sigma_2 & \textit{disjunction} \\ & & \sigma_1 \Longrightarrow \sigma_2 & \textit{implication} \\ & & \neg\sigma & \textit{negation} \end{array}$$

Fig. 8. Logic Formulas

$$\frac{}{c\ \mathsf{Term}}\ \textsc{Term}_1 \qquad \frac{}{x\ \mathsf{Term}}\ \textsc{Term}_2 \qquad \frac{t_1\ \mathsf{Term} \quad t_2\ \mathsf{Term} \quad \ldots \quad t_n\ \mathsf{Term}}{\{\ell_1 = t_1, \ldots, \ell_n = t_n\}\ \mathsf{Term}}\ \textsc{Term}_3 \qquad \frac{t\ \mathsf{Term}}{t \cdot \ell_i\ \mathsf{Term}}\ \textsc{Term}_4$$

$$\frac{t\ \mathsf{Term}}{i \cdot t\ \mathsf{Term}}\ \textsc{Term}_5 \qquad \frac{}{i\ \mathsf{Term}}\ \textsc{Term}_6 \qquad \frac{}{\_\ \mathsf{Term}}\ \textsc{Term}_7 \qquad \frac{t\ \mathsf{Term}}{x\ \mathsf{as}\ t\ \mathsf{Term}}\ \textsc{Term}_8 \qquad \frac{}{o\ \mathsf{Term}}\ \textsc{Term}_9$$

$$\frac{t\ \mathsf{Term}}{o\ t\ \mathsf{Term}}\ \textsc{Term}_{10}$$

Fig. 9. Term Judgment

Figure 8 defines the form of the formulas generated by our approach. The leaves of these formulas are equalities between base terms $t$, defined in Figure 9. These base terms encode three things: LambdaPix values, patterns, and the application of a primitive operation and a value. Encoding all of these things as terms allows a term equivalence to state that either two values are the same, that a value matches with a pattern, or that a primitive operation application yields a value that is equal to another value or matches with a pattern.

The inclusion of primitive operation applications as base terms is somewhat strange since they are not values in the actual dynamics of LambdaPix, but this inclusion enables the resulting formula to include all of the information pertaining to the theory from which the primitive operation originates. For instance, since the theory of quantifier-free linear integer arithmetic knows that addition is commutative, this inclusion makes it possible for the expressions $\lambda x.\lambda y.(x + y)$ and $\lambda x.\lambda y.(y + x)$ to be recognized as equivalent.

Except when a variable, primitive operation, as pattern, or wildcard pattern is included in one of the terms, term equivalence is identical to syntactic equality. When a primitive operation is included in a term, the specific primitive operation is used to determine how to understand the term equivalence (e.g. $1 + 2 \equiv 3$ is a valid term equivalence using the primitive operation "+"). When an as pattern is included in a term equivalence: $x\ \mathsf{as}\ e_1 \equiv e_2$, the term equivalence is the same as $x \equiv e_1 \wedge e_1 \equiv e_2$. When a wildcard pattern is included in a term equivalence: $\_ \equiv e$, the term equivalence can simply be interpreted as "true".

When one or more free variables are included in a formula, they must be resolved to determine the formula's truth. Throughout our approach, contexts are used to keep track of the types of all of a formula's free variables. Expressions can be substituted for variables of the same type in a formula to resolve it (e.g. $[3/x](x \equiv 1 \land x \equiv 2)$ yields $3 \equiv 1 \land 3 \equiv 2$). A formula is valid if it is true under all possible substitutions of its variables. To denote this, we define a new form of judgment:

*Definition 4.1* ($\overset{\text{val}}{\forall}_\Gamma.j$). If $\Gamma = \vec{x} : \vec{\tau}$, then the judgement $\overset{\text{val}}{\forall}_\Gamma.j$ 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. Implicitly, although the types of primitive operations are included in $\Gamma_{\text{initial}}$, and therefore $\Gamma$, we omit typings of the form $o : \tau_1 \longrightarrow \tau_2$ from $\vec{x} : \vec{\tau}$ so that we do not range over all possible meanings for LambdaPix's primitive operations. Then if $\Gamma$ is a typing context with a mapping for every free variable in a formula $\sigma$, the validity of $\sigma$ is denoted $\overset{\text{val}}{\forall}_\Gamma.\sigma$.

The validity of formulas will be what determines whether our approach recognizes two LambdaPix expressions as equivalent. Our approach takes as input two LambdaPix expressions and uses them to output a logic formula. In Section 6, we show that if the output formula is valid by Definition 4.1, then the two expressions are necessarily equivalent. To define our approach's method of constructing the logic formula from the original LambdaPix expressions in Section 4.4, we begin by first defining a few helper judgments pertaining to weak head reduction and freshening.

## 4.2 Weak Head Reduction $e \downarrow e'$

We do not have the option of fully evaluating the expressions during execution, as expressions may contain free variables in redex positions. For this reason we use weak head reduction at each step; this eliminates head-position redexes until free variables get in the way. The result is a weak head normal form expression.

$$\frac{e \rightsquigarrow e' \quad e' \downarrow e''}{e \downarrow e''}\ \textsc{BigWhnf}_1 \qquad \frac{e \not\rightsquigarrow}{e \downarrow e}\ \textsc{BigWhnf}_2 \qquad \frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2}\ \textsc{Whnf}_1$$

$$\frac{}{(\lambda x.e_1)e_2 \rightsquigarrow [e_2/x]e_1}\ \textsc{Whnf}_2 \qquad \frac{e \rightsquigarrow e'}{e \cdot \ell_i \rightsquigarrow e' \cdot \ell_i}\ \textsc{Whnf}_3 \qquad \frac{}{\{\ldots, \ell_i = e, \ldots\} \cdot \ell_i \rightsquigarrow e}\ \textsc{Whnf}_4$$

Fig. 10. Weak Head Reduction

## 4.3 Freshening

It is sometimes useful to generate fresh variables (globally unique variables) to avoid variable capture. As single variables are not the only form of binding sites in LamdbaPix, we generalize this notion to patterns. When freshen $p.e \hookrightarrow p'.e'$, $p'.e'$ is the same as $p.e$ except all variables bound by $p$ are alpha-varied to fresh variables. The definition of the freshen judgment is given in Figure 11.

In addition to creating fresh variables to avoid variable capture, our approach also sometimes generates fresh variables in order to couple the binding sites between two expressions being considered. For instance, if our approach knows that the same expression $e$ is being matched to variable $x$ in one expression and variable $y$ in another expression, it is useful to equate these bindings so that as our approach proceeds, it is able to know that $x$ in the first expression is the same as $y$ in the second expression. The judgment $\mathsf{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p.e_1', p.e_2')$ defined in Figure 12 does exactly that, taking in two bindings and returning freshened versions of those bindings

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

$$\frac{\text{freshen } p_1.e \hookrightarrow p_1'.e_1 \quad \text{freshen } p_2.e_1 \hookrightarrow p_2'.e_2 \quad \ldots \quad \text{freshen } p_n.e_{n-1} \hookrightarrow p_n'.e_n}{\text{freshen } \{\ell_1 = p_1, \ldots, \ell_n = p_n\}.e \hookrightarrow \{\ell_1 = p_1', \ldots, \ell_n = p_n'\}.e_n}\ \textsc{Freshen}_3$$

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

$$\frac{}{\text{freshen } i.e \hookrightarrow i.e}\ \textsc{Freshen}_6 \qquad \frac{\text{freshen } p.e \hookrightarrow p'.e'}{\text{freshen } i \cdot p.e \hookrightarrow i \cdot p'.e'}\ \textsc{Freshen}_7$$

Fig. 11. Freshening

$$\frac{}{\text{EB}(\_.e_1, \_.e_2) \hookrightarrow (\_.e_1, \_.e_2)}\ \text{EB}_1 \qquad \frac{y \text{ fresh}}{\text{EB}(\_.e_1, x.e_2) \hookrightarrow (y.e_1, y.[y/x]e_2)}\ \text{EB}_2$$

$$\frac{y \text{ fresh}}{\text{EB}(x.e_1, \_.e_2) \hookrightarrow (y.[y/x]e_1, y.e_2)}\ \text{EB}_3 \qquad \frac{y \text{ fresh}}{\text{EB}(x.e_1, x'.e_2) \hookrightarrow (y.[y/x]e_1, y.[y/x']e_2)}\ \text{EB}_4$$

$$\frac{y \text{ fresh} \quad \text{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p'.e_1', p'.e_2')}{\text{EB}(x \text{ as } p_1.e_1, p_2.e_2) \hookrightarrow (y \text{ as } p'.[y/x]e_1', y \text{ as } p'.e_2')}\ \text{EB}_5$$

$$\frac{y \text{ fresh} \quad \text{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p'.e_1', p'.e_2')}{\text{EB}(p_1.e_1, x \text{ as } p_2.e_2) \hookrightarrow (y \text{ as } p'.e_1', y \text{ as } p'.[y/x]e_2')}\ \text{EB}_6$$

$$\frac{\text{EB}(p_1.e_1, p_1'.e_2) \hookrightarrow (p_1''.e_1^1, p_1''.e_2^1) \quad \ldots \quad \text{EB}(p_n.e_1^{n-1}, p_n'.e_2^{n-1}) \hookrightarrow (p_n''.e_1^n, p_n''.e_2^n)}{\text{EB}(\{\ell_1 = p_1, \ldots \ell_n = p_n\}.e_1, \{\ell_1 = p_1', \ldots \ell_n = p_n'\}.e_2) \hookrightarrow (\{\ell_1 = p_1'', \ldots \ell_n = p_n''\}.e_1^n, \{\ell_1 = p_1'', \ldots \ell_n = p_n''\}.e_2^n)}\ \text{EB}_7$$

$$\frac{}{\text{EB}(c.e_1, c.e_2) \hookrightarrow (c.e_1, c.e_2)}\ \text{EB}_8 \qquad \frac{}{\text{EB}(i.e_1, i.e_2) \hookrightarrow (i.e_1, i.e_2)}\ \text{EB}_9$$

$$\frac{\text{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p'.e_1', p'.e_2')}{\text{EB}(i \cdot p_1.e_1, i \cdot p_2.e_2) \hookrightarrow (i \cdot p'.e_1', i \cdot p'.e_2')}\ \text{EB}_{10}$$

Fig. 12. Equate Bindings Judgment

that use the same variables so long as the two bindings $p_1.e_1$ and $p_2.e_2$ can be alpha-varied to use a shared pattern $p$.

The benefit of the equate bindings judgment specifically comes into play when comparing case expressions. If two case expressions are casing on the same $e$, and they have identical or near identical binding structures, then it is sometimes useful to freshen the case expressions together, so that as our approach proceeds to consider all of the possible outcomes of the case expressions, it is able to know that the same $e$ was bound in the same way in both expressions. The judgment $\text{FT}(\{p_1.e_1 \mid \ldots \mid p_n.e_n\}, \{p_1'.e_1' \mid \ldots \mid p_m'.e_m'\}) \overset{s}{\hookrightarrow} (\{p_1''.e_1'' \mid \ldots \mid p_n''.e_n''\}, \{p_1'''.e_1''' \mid \ldots \mid p_n'''.e_n'''\})$ defined in Figure 13 takes in two lists of bindings from case expressions, and equates the first $s$ bindings, independently freshening the rest. The judgment is defined so that once a pair of bindings

$$\frac{\mathsf{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p.e_1', p.e_2')}{\mathsf{FT}(\{p_1.e_1 \mid \cdot\}, \{p_2.e_2 \mid \cdot\}) \overset{1}{\hookrightarrow} (\{p.e_1' \mid \cdot\}, \{p.e_2' \mid \cdot\})}\ \mathrm{FT}_1$$

$$\frac{\mathsf{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p.e_1', p.e_2') \quad \mathsf{FT}(rest_1, rest_2) \overset{n}{\hookrightarrow} (rest_1', rest_2')}{\mathsf{FT}(\{p_1.e_1 \mid rest_1\}, \{p_2.e_2 \mid rest_2\}) \xhookrightarrow{n+1} (\{p.e_1' \mid rest_1'\}, \{p.e_2' \mid rest_2'\})}\ \mathrm{FT}_2$$

$$\frac{\forall_{i\in[n]}(\text{freshen } p_i.e_i \hookrightarrow p_i''.e_i'') \quad \forall_{i\in[m]}(\text{freshen } p_i'.e_i' \hookrightarrow p_i'''.e_i''')}{\mathsf{FT}(\{p_1.e_1 \mid \ldots \mid p_n.e_n\}, \{p_1'.e_1' \mid \ldots \mid p_m'.e_m'\}) \overset{0}{\hookrightarrow} (\{p_1''.e_1'' \mid \ldots \mid p_n''.e_n''\}, \{p_1'''.e_1''' \mid \ldots \mid p_m'''.e_m'''\})}\ \mathrm{FT}_3$$

Fig. 13. Freshen Together Judgment

cannot be equated, all subsequent bindings are freshened independently. This is done to ensure that no bindings are unsoundly equated. The rules listed in Figure 13 are listed in order of precedence (i.e. if it is possible to apply $\mathrm{FT}_2$ or $\mathrm{FT}_3$, it will apply $\mathrm{FT}_2$).

### 4.4 Formula Generation $\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \dashv \Gamma'$

The judgment that connects the validity of logic formulas with the equivalence of LambdaPix expressions is $\Gamma \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'$. The judgment that defines how our approach generates said logic formulas is $\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \dashv \Gamma'$.

When $\Gamma \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'$ or $\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \dashv \Gamma'$, the only free variables appearing in $e_1$ and $e_2$ are in $\Gamma$, so $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. However, $\sigma$ can contain more free variables than just those in $\Gamma$. The purpose of $\Gamma'$ is to describe the rest of the variables in $\sigma$. $\Gamma$ and $\Gamma'$ are disjoint and between them account for all variables which may appear in $\sigma$.

$$\frac{e_1 \downarrow e_1' \quad e_2 \downarrow e_2' \quad \Gamma \vdash e_1' \overset{\sigma}{\longleftrightarrow} e_2' : \tau \dashv \Gamma';}{\Gamma \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'}\ \textsc{IsoExp}$$

Fig. 14. IsoExp Rule

The judgment $\Gamma \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'$ is defined by Figure 14 and is mutually recursive with $\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \dashv \Gamma'$. We use it to define what it means for two expressions to be isomorphic.

*Definition 4.2 (Isomorphic).* We call two expressions $e_1$ and $e_2$ where $\Gamma_{\text{initial}} \vdash e_1 : \tau$ and $\Gamma_{\text{initial}} \vdash e_2 : \tau$ isomorphic if $\Gamma_{\text{initial}} \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'$ and $\overset{\text{val}}{\forall}_{\Gamma'}.\sigma$.

The purpose of the distinction between the two judgments is to allow our approach to perform weak head reduction exactly when needed. The judgment $\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \dashv \Gamma'$ assumes as a precondition that $e_1$ and $e_2$ are in weak head normal form, and is defined by Figures 15, 16, and 17.

Each rule in Figure 15 is written to address a particular syntactic form that $e_1$ and $e_2$ might take. Since each rule targets a particular syntactic form, the premises of each rule are motivated by the semantics of that form. For example, $\textsc{Iso}_{\text{lambda}}$ has the premises $x$ fresh and $\Gamma, x : \tau \vdash [x/x_1]e_1 \overset{\sigma}{\Longleftrightarrow} [x/x_2]e_2 : \tau' \dashv \Gamma'$. The former premise simply declares $x$ as a previously unused

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad e_1\ \mathsf{Term} \quad e_2\ \mathsf{Term}}{\Gamma \vdash e_1 \xleftrightarrow{e_1 \equiv e_2} e_2 : \tau \dashv \cdot}\ \textsc{Iso}_{\text{atomic}}$$

$$\frac{\Gamma \vdash e_1 \overset{\sigma_1}{\Longleftrightarrow} e_1' : \tau_1 \dashv \Gamma_1' \quad \ldots \quad \Gamma \vdash e_n \overset{\sigma_n}{\Longleftrightarrow} e_n' : \tau_n \dashv \Gamma_n'}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \xleftrightarrow{\sigma_1 \land \ldots \land \sigma_n} \{\ell_1 = e_1', \ldots, \ell_n = e_n'\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \dashv \Gamma_1', \ldots, \Gamma_n'}\ \textsc{Iso}_{\text{record}}$$

$$\frac{\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \{\ldots, \ell_i : \tau_i, \ldots\} \dashv \Gamma'}{\Gamma \vdash e_1 \cdot \ell_i \overset{\sigma}{\longleftrightarrow} e_2 \cdot \ell_i : \tau_i \dashv \Gamma'}\ \textsc{Iso}_{\text{projection}} \qquad \frac{i : \tau \hookrightarrow \delta \quad \Gamma \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'}{\Gamma \vdash i \cdot e_1 \overset{\sigma}{\longleftrightarrow} i \cdot e_2 : \delta \dashv \Gamma'}\ \textsc{Iso}_{\text{injection}}$$

$$\frac{x\ \mathsf{fresh} \quad \Gamma, x : \tau \vdash [x/x_1]e_1 \overset{\sigma}{\Longleftrightarrow} [x/x_2]e_2 : \tau' \dashv \Gamma'}{\Gamma \vdash \lambda x_1.e_1 \overset{\sigma}{\longleftrightarrow} \lambda x_2.e_2 : \tau \to \tau' \dashv x : \tau, \Gamma'}\ \textsc{Iso}_{\text{lambda}}$$

$$\frac{x\ \mathsf{fresh} \quad \Gamma, x : \tau \vdash [x/x_1]e_1 \overset{\sigma}{\Longleftrightarrow} [x/x_2]e_2 : \tau \dashv \Gamma'}{\Gamma \vdash \mathtt{fix}\ x_1\ \mathtt{is}\ e_1 \overset{\sigma}{\longleftrightarrow} \mathtt{fix}\ x_2\ \mathtt{is}\ e_2 : \tau \dashv x : \tau, \Gamma'}\ \textsc{Iso}_{\text{fix}}$$

Fig. 15. Formula Generation Rules

variable, and the latter premise states that if any value $x$ of type $\tau$ (the input type to both expressions) is substituted for $x_1$ in the left expression and $x_2$ in the right expression, then the two expressions will be equivalent if $\sigma$ is valid. This reflects the fact that two functions are equivalent if and only if their outputs are equivalent for all valid inputs.

Although the soundness of these rules is guaranteed, their completeness is not. For instance, $\textsc{Iso}_{\text{projection}}$ has the premise $\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \{\ldots, \ell_i : \tau_i, \ldots\} \dashv \Gamma'$. If this premise holds, then the conclusion that $\Gamma \vdash e_1 \cdot \ell_i \overset{\sigma}{\longleftrightarrow} e_2 \cdot \ell_i : \tau_i \dashv \Gamma'$ necessarily follows, as if two records are equivalent, then each of the records' respective entries must also be equivalent. But it is not the case that in order for two projections to be equivalent, they must project from equivalent records.

Each rule in Figure 16 addresses the case in which at least one of the expressions being compared is an application. When the two expressions being compared are both applications of equivalent arguments onto equivalent functions, $\textsc{Iso}_{\text{application1}}$ can be used to infer equivalence of the resulting applications. For situations in which an application is being compared to another syntactic form, or two applications that cannot be recognized as equivalent via $\textsc{Iso}_{\text{application1}}$ are being compared, the remaining rules take an application and replace it with a shared fresh variable in both expressions. For example, if the expressions $f(x)$ and $f(x+0)$ are being compared, $\textsc{Iso}_{\text{application1}}$ is sufficient to find equivalence because $f$ can be found equivalent to $f$ and $x$ can be found equivalent to $x+0$ via $\textsc{Iso}_{\text{atomic}}$. But if $f(x)$ and $f(x)+0$ are being compared, $\textsc{Iso}_{\text{application1}}$ alone would be insufficient, as the outermost function of the first expression is $f$ and the outermost function of the second expression is $+$. For this situation, $\textsc{Iso}_{\text{application2}}$ would be needed to replace $f(x)$ with the fresh variable $y$, yielding the expressions $y$ and $y+0$, which can be immediately found equivalent via $\textsc{Iso}_{\text{atomic}}$.

The current formula generation application rules have multiple limitations. First, the rules only allow applications to be replaced with shared fresh variables when the application being replaced is at the outermost level of one of the expressions. This has the consequence that although $f(x)+f(x)$ and $2 * f(x)$ are obviously equivalent, and the substitution of $f(x)$ for a shared fresh

$$\frac{\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \rightarrow \tau' \dashv \Gamma' \quad \Gamma \vdash e_1' \overset{\sigma'}{\Longleftrightarrow} e_2' : \tau \dashv \Gamma''}{\Gamma \vdash e_1\ e_1' \overset{\sigma \wedge \sigma'}{\longleftrightarrow} e_2\ e_2' : \tau' \dashv \Gamma', \Gamma''}\ \text{Iso}_{\text{application1}}$$

$$\frac{y \text{ fresh} \quad \Gamma, y : \tau \vdash y \overset{\sigma}{\longleftrightarrow} [y/(x\ e_1)]e_2 : \tau \dashv \Gamma'}{\Gamma \vdash x\ e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \dashv \Gamma'}\ \text{Iso}_{\text{application2}}$$

$$\frac{y \text{ fresh} \quad \Gamma, y : \tau \vdash [y/(x\ e_2)]e_1 \overset{\sigma}{\longleftrightarrow} y : \tau \dashv \Gamma'}{\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} x\ e_2 : \tau \dashv \Gamma'}\ \text{Iso}_{\text{application3}}$$

$$\frac{y \text{ fresh} \quad \Gamma, y : \tau \vdash y \overset{\sigma}{\longleftrightarrow} [y/(o\ e_1)]e_2 : \tau \dashv \Gamma'}{\Gamma \vdash o\ e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \dashv \Gamma'}\ \text{Iso}_{\text{application4}}$$

$$\frac{y \text{ fresh} \quad \Gamma, y : \tau \vdash [y/(o\ e_2)]e_1 \overset{\sigma}{\longleftrightarrow} y : \tau \dashv \Gamma'}{\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} o\ e_2 : \tau \dashv \Gamma'}\ \text{Iso}_{\text{application5}}$$

$$\frac{y \text{ fresh} \quad \Gamma, y : \tau \vdash y \overset{\sigma}{\longleftrightarrow} [y/((\texttt{fix}\ x_1\ \texttt{is}\ e_1)\ e_2)]e : \tau \dashv \Gamma'}{\Gamma \vdash (\texttt{fix}\ x_1\ \texttt{is}\ e_1)\ e_2 \overset{\sigma}{\longleftrightarrow} e : \tau \dashv \Gamma'}\ \text{Iso}_{\text{application6}}$$

$$\frac{y \text{ fresh} \quad \Gamma, y : \tau \vdash [y/((\texttt{fix}\ x_1\ \texttt{is}\ e_1)\ e_2)]e \overset{\sigma}{\longleftrightarrow} y : \tau \dashv \Gamma'}{\Gamma \vdash e \overset{\sigma}{\longleftrightarrow} (\texttt{fix}\ x_1\ \texttt{is}\ e_1)\ e_2 : \tau \dashv \Gamma'}\ \text{Iso}_{\text{application7}}$$

Fig. 16. Formula Generation Application Rules

variable $y$ would enable $\text{Iso}_{\text{atomic}}$ to prove that fact, our current rules do not support this inference. Second, $\text{Iso}_{\text{application6}}$ and $\text{Iso}_{\text{application7}}$ require substituting an entire fixed point application in an expression, so unless if the two expressions being compared have essentially identical fixed points included, these rules will be ineffective. Still, despite these limitations, the current formula generation application rules are sufficient for their most common purpose of working with $\text{Iso}_{\text{fix}}$ to ensure that recursive function calls are recognized as equivalent when given equivalent arguments.

Each rule in Figure 17 addresses the situation in which at least one of the expressions being compared is a case analysis. These rules can be grouped into two broad approaches. For situations in which only one of the expressions being compared is a case analysis, or both expressions are case analyses but the expressions being cased on are not equivalent, $\text{Iso}_{\text{case1}}$ and $\text{Iso}_{\text{case2}}$ are used to unpack one case analysis at a time. If $\texttt{case}\ e\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\}$ is being compared to $e'$, then the formula generated by these rules essentially states that if $e$ can be pattern matched with $p_i$ and no prior patterns, $e_i$ needs to be equivalent to $e'$ in order for the two overall expressions to be equivalent.

For situations in which the two expressions being compared are case analyses that are casing on equivalent expressions, $\text{Iso}_{\text{case3}}$, $\text{Iso}_{\text{case4}}$, and $\text{Iso}_{\text{case5}}$ are used to deconstruct both case expressions simultaneously. To do this, $\text{Iso}_{\text{case3}}$ is always used first to ensure that the expressions being cased on are equivalent. If the expressions being cased on are not equivalent, then $\sigma$ in the formula generated by $\text{Iso}_{\text{case3}}$ will not be valid, and so the output formula $\sigma \wedge \sigma'$ will not be valid as a result. If the expressions being cased on are equivalent, then $\text{Iso}_{\text{case4}}$ and $\text{Iso}_{\text{case5}}$ can be used

$$\frac{e\ \mathsf{Term} \quad \forall_{i\in[n]}\left(\mathsf{freshen}\ p_i.e_i \hookrightarrow p_i'.e_i' \quad p_i' :: \tau' \dashv \Gamma_i \quad \Gamma,\Gamma_i \vdash e_i' \overset{\sigma_i}{\Longleftrightarrow} e' : \tau \dashv \Gamma_i'\right)}{\Gamma \vdash \mathsf{case}\ e\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\} \xleftrightarrow{\wedge_{i\in[n]}((\wedge_{j\in[i-1]}(e \not\equiv p_j')) \wedge e \equiv p_i') \Rightarrow \sigma_i} e' : \tau \dashv \forall_{i\in[n]}\Gamma_i,\Gamma_i'}\ \mathrm{Iso}_{\mathrm{case1}}$$

$$\frac{e\ \mathsf{Term} \quad \forall_{i\in[n]}\left(\mathsf{freshen}\ p_i.e_i \hookrightarrow p_i'.e_i' \quad p_i' :: \tau' \dashv \Gamma_i \quad \Gamma,\Gamma_i \vdash e_i' \overset{\sigma_i}{\Longleftrightarrow} e' : \tau \dashv \Gamma_i'\right)}{\Gamma \vdash e' \xleftrightarrow{\wedge_{i\in[n]}((\wedge_{j\in[i-1]}(e \not\equiv p_j')) \wedge e \equiv p_i') \Rightarrow \sigma_i} \mathsf{case}\ e\ \{p_1.e_1 \mid \ldots \mid p_n.e_n\} : \tau \dashv \forall_{i\in[n]}\Gamma_i,\Gamma_i'}\ \mathrm{Iso}_{\mathrm{case2}}$$

$$\frac{\Gamma \vdash e \overset{\sigma}{\longleftrightarrow} e' : \tau' \dashv \Gamma' \quad x\ \mathsf{fresh} \quad \Gamma, x : \tau' \vdash \mathsf{case}\ x\ \{\ldots\} \overset{\sigma'}{\longleftrightarrow} \mathsf{case}\ x\ \{\ldots'\} : \tau \dashv \Gamma''}{\Gamma \vdash \mathsf{case}\ e\ \{\ldots\} \overset{\sigma\wedge\sigma'}{\longleftrightarrow} \mathsf{case}\ e'\ \{\ldots'\} : \tau \dashv \Gamma', x : \tau', \Gamma''}\ \mathrm{Iso}_{\mathrm{case3}}$$

$$\frac{\begin{array}{c}\mathsf{FT}(\{M\},\{M'\}) \overset{s}{\hookrightarrow} (\{p_1.e_1 \mid \ldots \mid p_n.e_n\},\{p_1'.e_1' \mid \ldots \mid p_m'.e_m'\}) \\ \forall_{i\in[s]}(p_i :: \tau' \dashv \Gamma_i \quad \Gamma,\Gamma_i \vdash e_i \overset{\sigma_i}{\Longleftrightarrow} e_i' \dashv \Gamma_i') \\ \forall_{j\in[s+1,n]}(p_j :: \tau' \dashv \Gamma_j \quad \Gamma,\Gamma_j \vdash e_j \overset{\sigma_j}{\Longleftrightarrow} \mathsf{case}\ x\ \{p_1'.e_1' \mid \ldots \mid p_m'.e_m'\} : \tau \dashv \Gamma_j')\end{array}}{\Gamma \vdash \mathsf{case}\ x\ \{M\} \overset{\Psi}{\longleftrightarrow} \mathsf{case}\ x\ \{M'\} : \tau \dashv \forall_{i\in[n]}\Gamma_i,\Gamma_i'}\ \mathrm{Iso}_{\mathrm{case4}}$$

$$\frac{\begin{array}{c}\mathsf{FT}(\{M'\},\{M\}) \overset{s}{\hookrightarrow} (\{p_1'.e_1' \mid \ldots \mid p_m'.e_m'\},\{p_1.e_1 \mid \ldots \mid p_n.e_n\}) \\ \forall_{i\in[s]}(p_i :: \tau' \dashv \Gamma_i \quad \Gamma,\Gamma_i \vdash e_i \overset{\sigma_i}{\Longleftrightarrow} e_i' \dashv \Gamma_i') \\ \forall_{j\in[s+1,n]}(p_j :: \tau' \dashv \Gamma_j \quad \Gamma,\Gamma_j \vdash \mathsf{case}\ x\ \{p_1'.e_1' \mid \ldots \mid p_m'.e_m'\} \overset{\sigma_j}{\Longleftrightarrow} e_j : \tau \dashv \Gamma_j')\end{array}}{\Gamma \vdash \mathsf{case}\ x\ \{M'\} \overset{\Psi}{\longleftrightarrow} \mathsf{case}\ x\ \{M\} : \tau \dashv \forall_{i\in[n]}\Gamma_i,\Gamma_i'}\ \mathrm{Iso}_{\mathrm{case5}}$$

$$\Psi := (\wedge_{i\in[s]}\sigma_i) \wedge (\wedge_{j\in[s+1,n]}((\wedge_{k\in[j-1]}(x \not\equiv p_k)) \wedge x \equiv p_j) \Rightarrow \sigma_j)$$

Fig. 17. Formula Generation Case Rules

to generate $\sigma'$. This approach is needed in addition to $\mathrm{Iso}_{\mathrm{case1}}$ and $\mathrm{Iso}_{\mathrm{case2}}$ because $\mathrm{Iso}_{\mathrm{case1}}$ and $\mathrm{Iso}_{\mathrm{case2}}$ require that the expression being cased on is a base term.

All rules in Figure 15 are deterministic in the sense that for all possible expressions, at most one rule is applicable. However, the rules in Figures 16 and 17 are non-deterministic. If two case expressions or two applications are being compared, there may be multiple applicable rules. For instance, if case 1 {1.2|_.3} is being compared to case 2 {_.2}, then $\mathrm{Iso}_{\mathrm{case1}}$, $\mathrm{Iso}_{\mathrm{case2}}$, and $\mathrm{Iso}_{\mathrm{case3}}$ are all applicable. Our approach handles this by considering all formulas that can be generated by applying any applicable rule and outputs the disjunction of all generated formulas. We will later show that applying any applicable rule in such a situation is sound and that therefore, taking the disjunction of all generated formulas is also sound. The only exception to this is that $\mathrm{Iso}_{\mathrm{case3}}$ cannot be applied multiple times in a row because it is never useful to do so and allowing this would cause an infinite loop.

In instances where there is no applicable rule, such as if $i_1\ e_1$ is compared with $i_2\ e_2$ where $i_1 \neq i_2$ and either $e_1$ or $e_2$ cannot be encoded into a term, our approach simply outputs the formula $\sigma$=False, which is always sound.

### 4.5 Limitations and Further Extensions

The current set of rules is comprehensive and covers a wide range of operators that are often found in many functional programming assignments. However, there are limitations to the current set of rules, some of which have already been noted. The current main limitations include:

- Our current handling of projections in $\mathrm{Iso}_{\mathrm{projection}}$ requires that in order for two projections to be recognized as equivalent, they must project from equivalent records.
- Our current handling of recursive function calls occurs entirely through the interplay between $\mathrm{Iso}_{\mathrm{fix}}$ and the formula generation application rules. Because of how these rules are currently defined, recursive functions can only be recognized as equivalent if in all situations they recurse on equivalent arguments or do not recurse at all.
- The approach taken by the formula generation application rules is limited in that applications can only be replaced with shared fresh variables when the application being replaced is at the outermost level of one of the expressions being compared.
- $\mathrm{Iso}_{\mathrm{application6}}$ and $\mathrm{Iso}_{\mathrm{application7}}$ both require substituting a variable for an entire fixed point application, which will only be useful if the two expressions being compared have essentially identical fixed points included.
- Since $\mathrm{Iso}_{\mathrm{case1}}$ and $\mathrm{Iso}_{\mathrm{case2}}$ require that the expression being cased on is a base term, the current set of rules cannot identify equivalence between a case analysis in which the expression being cased on isn't a base term and any other syntactic form.
- The current definition of LambdaPix does not allow for state, and so our approach cannot identify the equivalence of any programs that use state.

Of the various extensions that could be implemented to address an aforementioned limitation, extending LambdaPix to support state would likely require a significant number of changes to our approach. However, this could be potentially achieved by handling sequential state-altering declaration similar to how we handle local declaration. Currently, we handle local declaration by encoding the declaration into the SMT formula in the same way that we would encode a single pattern case expression (i.e. `let val x =` $e_1$ `in` $e_2$ `end` becomes `case` $e_1$ $\{x.e_2\}$ at the transpilation to LambdaPix stage). It would not be possible to do the same procedure for sequential declaration since the scoping would have to be global. However, we believe that it may be feasible to treat reference assignment and update similar to variable declaration and shadowing with modified scoping.

One advantage of the structure of our approach is that extending our system to address some of the previously listed limitations is straightforward. As soon as a new rule that addresses one of the system's current limitations is found to be sound, it can be simply tacked on to the current system without needing to modify any preexisting rules. This also applies to extensions of the underlying language LambdaPix itself. Adding new base types to LambdaPix such as strings or reals requires no modification of the current rules whatsoever, and adding additional syntactic expression forms requires only the addition of rules for comparing the new form against itself and arbitrary expressions. Even though it is easy to extend the LambdaPix language and add additional rules, the current version is already rich enough to capture common behavior in programming assignments of introductory courses.

## 5 OPERATION

To provide a better understanding of our approach, we step through our approach's operation on a pair of simple Standard ML expressions provided above. As we step through this example, we will refer to the inference rules from the previous section to illustrate how they are applied.

```
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)
  ))
```

First, we transpile both expressions to LambdaPix. This is shown above. Since much of the proof derivation which drives our approach is free of branching, through most of this section we will view our approach as transforming the above expressions through the application of rules, rather than building up a proof tree.

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

```
λx.λy.
  (λa.λf.
    case a of
    { SOME·b.f b
    | NONE.NONE }
  ) x (λm.
  (λa.λf.
    case a of
    { SOME·b.f b
    | NONE.NONE }
  ) y (λn.
    (λe.SOME·e) (m+n)
  ))
```

The entry point to our approach is the $\Gamma \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'$ judgement, defined by the rule IsoExp. By this rule, we reduce both expressions to weak head normal form then apply the $\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \dashv \Gamma'$ judgement to them. However, since the expressions in consideration are abstractions, the expressions are already in weak head normal form, so no transformation is necessary to apply this rule.

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

```
(λa.λf.
  case a of
  { SOME·b.f b
  | NONE.NONE }
) x (λm.
 (λa.λf.
   case a of
   { SOME·b.f b
   | NONE.NONE }
 ) y (λn.
   (λe.SOME·e) (m+n)
))
```

Next, since both expressions are lambda expressions with two curried arguments, we proceed with two applications of the rule $\text{Iso}_{\text{lambda}}$ . This requires us to create two new fresh variables and substitute them for the first two function arguments in both expressions. For simplicity, we will simply call the first fresh variable x and the second fresh variable y even though these names conflict with the original variable names. The key difference between before and after this process is that before this process, the two functions had the same variable names x and y by coincidence, whereas after this process, the two functions use the same fresh variables x and y by design. These applications of $\text{Iso}_{\text{lambda}}$ yield the above expressions.

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

```
case x of
{ SOME·b.
   (λm.
    (λa.λf.
     case a of
     { SOME·b.f b
     | NONE. NONE }
    ) y (λn.(λe.SOME·e) (m+n))
   ) b
| NONE.NONE }
```

Since the premise of $\text{Iso}_{\text{lambda}}$ invokes the ⇔ judgement, IsoExp requires that we reduce both expressions to weak head normal form. The left expression is already in weak head normal form, so no transformation is necessary, but the right expression must undergo two beta reductions before it is in weak head normal form. The result of these beta reductions is above.

Since both expressions are case expressions, our approach has multiple options for how to proceed. Formally, our approach pursues all of these options, generating separate formulas for each option, and finally outputting a disjunction of all of the generated formulas. This ensures that if any option can generate a valid formula, then the final result will be the disjunction of the valid formula with several other formulas, which altogether is valid. In this case, attempting to proceed with $\text{Iso}_{\text{case3}}$ will not yield a valid formula because the two expressions are casing on different things, but applying either $\text{Iso}_{\text{case1}}$ or $\text{Iso}_{\text{case2}}$ can yield a valid formula. For this demonstration, we step through the derivation that results from applying $\text{Iso}_{\text{case1}}$ and call the formulas generated by applying $\text{Iso}_{\text{case2}}$ or $\text{Iso}_{\text{case3}}$ $\sigma_{\text{Iso}_{\text{case2}}}$ and $\sigma_{\text{Iso}_{\text{case3}}}$ respectively.

As there are three branches in the left case expression, our approach's proof tree now splits into three branches. For this demonstration, we just step through the first of these branches, as the other two branches work similarly. We call the formulas generated by the other two branches of the proof tree $\sigma_{\text{branch 2}}$ and $\sigma_{\text{branch 3}}$.

```
SOME·(m1+n1)
```

```
case x of
{ SOME·b.
   (λm.
    (λa.λf.
     case a of
     { SOME·b.f b
     | NONE. NONE }
    ) y (λn.(λe.SOME·e) (m+n))
   ) b
| NONE.NONE }
```

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

$$((\mathtt{x},\mathtt{y}) \equiv (\mathtt{SOME{\cdot}m1},\mathtt{SOME{\cdot}n1})) \Rightarrow \ldots$$

where the ellipses is what we are going to fill in as we complete this branch of the proof tree.

Since the left expression has been simplified to a base term, our approach proceeds to work on the right expression. Our approach applies $\mathrm{Iso}_{\mathrm{case2}}$ twice (using beta reduction to reduce the expression to weak head normal form as appropriate), and finishes each branch of the proof tree by using $\mathrm{Iso}_{\mathrm{atomic}}$ to compare base terms.

Putting everything together, the final formula is:

$$(\sigma_{\text{branch 1}} \wedge \sigma_{\text{branch 2}} \wedge \sigma_{\text{branch 3}}) \vee \sigma_{\mathrm{Iso}_{\mathrm{case2}}} \vee \sigma_{\mathrm{Iso}_{\mathrm{case3}}}$$

where $\sigma_{\text{branch 1}}$ is

$$\begin{aligned}
&((\mathtt{x},\mathtt{y}) \equiv (\mathtt{SOME{\cdot}m1},\mathtt{SOME{\cdot}n1})) \Rightarrow\\
&\quad((\mathtt{x} \equiv \mathtt{SOME{\cdot}b1}) \Rightarrow\\
&\qquad(\mathtt{y} \equiv \mathtt{SOME{\cdot}b2}) \Rightarrow (\mathtt{SOME{\cdot}(m1+n1)} \equiv \mathtt{SOME{\cdot}(b1+b2)})\wedge\\
&\qquad(\mathtt{y} \not\equiv \mathtt{SOME{\cdot}b2} \wedge \mathtt{y} \equiv \mathtt{NONE}) \Rightarrow (\mathtt{SOME{\cdot}(m1+n1)} \equiv \mathtt{NONE})\\
&\quad)\wedge\\
&\quad((\mathtt{x} \not\equiv \mathtt{SOME{\cdot}b1} \wedge \mathtt{x} \equiv \mathtt{NONE}) \Rightarrow\\
&\qquad(\mathtt{y} \equiv \mathtt{SOME{\cdot}b2}) \Rightarrow (\mathtt{SOME{\cdot}(m1+n1)} \equiv \mathtt{NONE})\wedge\\
&\qquad(\mathtt{y} \not\equiv \mathtt{SOME{\cdot}b2} \wedge \mathtt{y} \equiv \mathtt{NONE}) \Rightarrow (\mathtt{SOME{\cdot}(m1+n1)} \equiv \mathtt{NONE})\\
&\quad)
\end{aligned}$$

and $\sigma_{\text{branch 2}}$ and $\sigma_{\text{branch 3}}$ are similar.

Since the two original expressions were equivalent, this formula is valid. The validity of this formula can be verified either by hand or by an SMT Solver.

## 6 SOUNDNESS

We prove the soundness of our approach: if our approach takes in two expressions and outputs a valid formula, then the two expressions must be equivalent.

### 6.1 Extensional Equivalence

To prove the soundness of our approach, we must first define what it means for two expressions to be equivalent. For this, we introduce extensional equivalence, a widely accepted notion of equivalence. Extensional equivalence is the same as contextual equivalence, and so two extensionally equivalent expressions are indistinguishable in terms of behavior. This implies that extensional equivalence is closed under evaluation. 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 $\tau$.

*Definition 6.1 (Extensional Equivalence).* We define that $e_1 \cong e_2 : \tau$ if $\Gamma_{\text{initial}} \vdash e_1 : \tau$, $\Gamma_{\text{initial}} \vdash e_2 : \tau$, $e_1 \Longmapsto v_1$, $e_2 \Longmapsto v_2$, and

(1) Rule $\mathrm{EQ}_1$: In the case that $\tau = \tau_1 \to \tau_2$, for all expressions $v$ such that $\Gamma_{\text{initial}} \vdash v : \tau_1$, $v_1\ v \cong v_2\ v : \tau_2$.
(2) Rule $\mathrm{EQ}_2$: In the case that $\tau$ is not an arrow type, for all patterns $p$ such that $p :: \tau$, either $v_1 \mathbin{/\!/} p \dashv B$ and $v_2 \mathbin{/\!/} p \dashv B$ or $v_1 \mathbin{\not/\!\!\not/} p$ and $v_2 \mathbin{\not/\!\!\not/} p$.

Unlike our approach, extensional equivalence inducts over the types of the expressions rather than their syntax, and is defined only over closed expressions. As we are only concerned with proving our approach sound over valuable expressions, we leave extensional equivalence undefined for divergent expressions.

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.

The soundness theorem for our approach connects our technique's definition of isomorphic with this definition of extensional equivalence. It is as follows:

Theorem 6.2 (Soundness). *For any expressions $e_1$ and $e_2$, if $\Gamma_{\text{initial}} \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'$ and $\overset{\text{val}}{\forall}_{\Gamma'}.\sigma$, then $e_1 \cong e_2 : \tau$.*

## 6.2 Proof Sketch

As the $\Gamma \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'$ judgement is defined simultaneously with the $\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} e_2 : \tau \dashv \Gamma'$ judgement, we prove the theorem by simultaneous induction on both of these judgements. We also use the $\overset{\text{val}}{\forall}_{\Gamma}.j$ judgement to strengthen the inductive hypotheses to account for variables. Recall that if $\Gamma = \vec{x} : \vec{\tau}$, then the judgement $\overset{\text{val}}{\forall}_{\Gamma}.j$ 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 (implicitly, we omit any primitive operations from the context $\Gamma = \vec{x} : \vec{\tau}$ as to not range over all possible meanings for LambdaPix's primitive operations). The theorem we wish to show by induction is then:

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

We first verify that the above statements imply the soundness theorem. Indeed, when $\Gamma_{\text{initial}} \vdash e_1 \overset{\sigma}{\Longleftrightarrow} e_2 : \tau \dashv \Gamma'$ we have $\overset{\text{val}}{\forall}_{\Gamma_{\text{initial}}}.\left(\text{if } \left(\overset{\text{val}}{\forall}_{\Gamma'}.\sigma\right) \text{ then } e_1 \cong e_2 : \tau\right)$. Since $\Gamma_{\text{initial}}$ contains only primitive operations, which are omitted from the $\overset{\text{val}}{\forall}_{\Gamma}.j$ judgment, the outer quantifier quantifies over no variables, so we have that if $\left(\overset{\text{val}}{\forall}_{\Gamma'}.\sigma\right)$ then $e_1 \cong e_2 : \tau$. This together with the assumption that $\overset{\text{val}}{\forall}_{\Gamma'}.\sigma$ allows us to conclude that $e_1 \cong e_2 : \tau$.

The full proof of each rule's soundness has 18 cases and uses 14 lemmas and can be found in the extended version of this paper [Clune et al. 2020]. Two cases are included below as examples:

$\text{Iso}_{\text{record}}$ : 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 $[\vec{v}/\vec{x}]\left(\overset{\text{val}}{\forall}_{\Gamma'_1,\ldots,\Gamma'_n}.\sigma_1 \wedge \ldots \sigma_n\right)$. It must be shown that $[\vec{v}/\vec{x}](\{\ell_1 = e_1, \ldots, \ell_n = e_n\} \cong \{\ell_1 = e'_1, \ldots, \ell_n = e'_n\})$.

Lemma 6.3. *If $e_1 \cong e_2 : \tau$, $e_1 \Longmapsto v_1$, and $e_2 \Longmapsto v_2$, then for all patterns $p$ where $p :: \tau \dashv \Gamma$, it is the case that either $v_1 /\!\!/ p \dashv B$ and $v_2 /\!\!/ p \dashv B$ or $v_1 \not\!/\!\!/ p$ and $v_2 \not\!/\!\!/ p$. Proof: by induction on $e_1 \cong e_2 : \tau$. If $\tau = \tau_1 \longrightarrow \tau_2$ then by inversion of $p :: \tau \dashv \Gamma$, $p$ must either be a wildcard or a variable. Then by $\text{Match}_1$ and $\text{Match}_2$, we have that $v_1 /\!\!/ p \dashv B$ and $v_2 /\!\!/ p \dashv B$. If $\tau$ is not an arrow type, then we conclude by $EQ_2$.*

By conjunction and that all the $\Gamma_i'$ are disjoint, we have that for all $i \in [n]$, $\overset{\text{val}}{\forall}_{\Gamma_i'}.\sigma_i$. Then by the inductive hypotheses, we have that $[\vec{v}/\vec{x}](e_i \cong e_i' : \tau_i)$. Since we are only concerned with proving our approach sound over valuable expressions, without loss of generality, we can assume that $[\vec{v}/\vec{x}]e_i \Longmapsto v_i$ and $[\vec{v}/\vec{x}]e_i' \Longmapsto v_i'$ for some values $v_i$ and $v_i'$. By Lemma 6.3, we have that for all $p_i$ where $p_i :: \tau_i \dashv \Gamma_i$, either $v_i \mathbin{/\!\!/} p_i \dashv B_i$ and $v_i' \mathbin{/\!\!/} p_i \dashv B_i$ or $v_i \not\mathbin{/\!\!/} p_i$ and $v_i' \not\mathbin{/\!\!/} p_i$.

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

- In the case that for all $i \in [n]$ $v_i \mathbin{/\!\!/} p_i \dashv B_i$ and $v_i' \mathbin{/\!\!/} p_i \dashv B_i$, by $\text{Match}_3$ we have $\{\ell_1 = v_1, \dots, \ell_n = v_n\} \mathbin{/\!\!/} p \dashv B_1 \dots B_n$ and $\{\ell_1 = v_1', \dots, \ell_n = v_n'\} \mathbin{/\!\!/} p \dashv B_1 \dots B_n$.
- In the case that there is some $i \in [n]$ where $v_i \not\mathbin{/\!\!/} p_i$ and $v_i' \not\mathbin{/\!\!/} p_i$, by $\text{Match}_4$ we have $\{\ell_1 = v_1, \dots, \ell_n = v_n\} \not\mathbin{/\!\!/} p$ and $\{\ell_1 = v_1', \dots, \ell_n = v_n'\} \not\mathbin{/\!\!/} p$.

Since in all cases either $\{\ell_1 = v_1, \dots, \ell_n = v_n\} \mathbin{/\!\!/} p \dashv B$ and $\{\ell_1 = v_1', \dots, \ell_n = v_n'\} \mathbin{/\!\!/} p \dashv B$ or $\{\ell_1 = v_1, \dots, \ell_n = v_n\} \not\mathbin{/\!\!/} p$ and $\{\ell_1 = v_1', \dots, \ell_n = v_n'\} \not\mathbin{/\!\!/} p$, by $\text{EQ}_2$, we may conclude

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

$\text{Iso}_{\text{application2}}$ : Let $\Gamma = \vec{z} : \vec{\tau}$ and let $\vec{v}$ be arbitrary where $v_i : \tau_i$ and $v_i$ val for all $v_i \in \vec{v}$. Assume $[\vec{v}/\vec{z}]\left(\overset{\text{val}}{\forall}_{\Gamma'}.\sigma\right)$. It must be shown that $[\vec{v}/\vec{z}](x\ e_1 \cong e_2)$.

By the inductive hypothesis we have

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

Since we are only concerned with proving our approach sound over valuable expressions, without loss of generality, we can assume that $x\ e_1 \Longmapsto w$ for some value $w$ such that $\Gamma \vdash w : \tau$ and $w$ val. Since $y$ is fresh, the inductive hypothesis written above implies

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

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

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

which is equivalent to

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

Since $x\ e_1 \Longmapsto w$, the two are extensionally equivalent. By the referential transparency of LambdaPix, the above expression is equivalent to

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

which is simply

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

## 7 EXPERIMENTAL RESULTS

We implemented our approach in a tool called ZEUS to serve as a grading assistant by clustering equivalent programs into equivalence classes. The goal of our evaluation is to answer the following questions:

Q1. Can ZEUS automatically identify equivalent programs in programming assignments for introductory functional programming courses?

Q2. How many equivalence classes are found by ZEUS?

Q3. What is the runtime performance of ZEUS?

### 7.1 Implementation

ZEUS is implemented in Standard ML and is publicly available as open-source at https://github.com/CMU-TOP/zeus. ZEUS takes as input a set of homework assignments from an introductory functional programming course at the college level taught in Standard ML. Each submission is transpiled from Standard ML into LambdaPix, and then ZEUS is run pairwise on the transpiled expressions and outputs a logical formula. If this formula is valid, then both expressions are algorithmically similar and guaranteed to be equivalent, so they are placed into the same equivalence class. As an optimization, since extensional equivalence is transitive, if ZEUS verifies that two programs $p_1$ and $p_2$ are (not) equivalent, and that $p_1$ is also (not) equivalent to $p_3$, then ZEUS does not check that $p_2$ is equivalent to $p_3$. This optimization significantly reduces the number of comparisons that otherwise would be quadratic in the number of assignments.

In the definition of LambdaPix, we assumed an arbitrary fixed set of disjoint algebraic datatypes with unique associated injection labels. This is unrealistic for an implementation since Standard ML includes datatype declarations. Our transpilation from Standard ML to LambdaPix instead scrapes all datatype declarations from the original Standard ML submission and uses those datatypes and their constructors as LambdaPix's set of datatypes and injection labels.

To determine the validity of the formulas generated by ZEUS, we use the SMT solver Z3 [de Moura and Bjørner 2008] using the theory of quantifier-free linear integer arithmetic and the theory of datatypes. From the theory of quantifier-free linear integer arithmetic, we use the built-in functions "$+$", "$-$", "$*$", "$\leq$", "$<$", "$\geq$", and "$>$", corresponding to the primitive operations of LambdaPix. We use the theory of datatypes to represent base terms of all types aside from *ints* and *booleans*.

Although we only use these two theories in ZEUS, nothing restricts a different implementation from using additional theories. For instance, another implementation could leverage the theory of *strings* by adding strings as a base type in LambdaPix and adding the SMT solver's built-in string functions to LambdaPix's set of primitive operations.

### 7.2 Benchmarks

To evaluate ZEUS, we used more than 4,000 student submissions from an introductory functional programming course. The number of submissions varies between 318 and 351 per assignment. Table 1 describes the twelve assignments that were used in our evaluation. These assignments show a large diversity of programs that includes different datatypes and the use of pattern matching and are a good test suite to test the applicability of ZEUS as a grading assistant. Figure 18 shows some of the datatype declarations that are assumed by the homework assignments presented in Table 1.

### 7.3 Clustering of equivalent programs

Tables 2 and 3 analyze the equivalent classes detected by ZEUS. In particular, for each task, Table 2 shows the number of submissions (#), the number of equivalent classes (ECs), the number of equivalence classes that contain 90% and 75% of the submissions (90th and 75th Percentile ECs,

Table 1. Description of homework assignments used in our evaluation

| Function | Signature | Description |
|---|---|---|
| concat | int list list → int list | concat takes a list of int lists and returns their concatenation without using the built-in "@" function |
| prefixSum | int list → int list | prefixSum replaces each *i-th* element in an int list with the sum of the list's first $i + 1$ elements |
| countNonZero | int tree → int | countNonZero takes an int tree T and returns the number of nonzero nodes in T |
| quicksort | ('a * 'a → order) * 'a list → 'a list | quicksort implements the quicksort algorithm |
| slowDoop | ('a * 'a → order) * 'a list → 'a list | slowDoop takes a comparison function and uses it to remove all duplicates in a list |
| differentiate | (int → real) → (int → real) | differentiate differentiates a polynomial that is represented with the type int → real |
| integrate | (int → real) → real → (int → real) | integrate takes a polynomial p and a real c and returns the antiderivative of p with constant of integration c |
| treefoldr | ('a * 'b → 'b) → 'b → 'a tree → 'b | (treefoldr g init T) returns (foldr g init L) where L is the inorder traversal of T |
| treeReduce | ('a * 'a → 'a) → 'a → 'a tree → 'a | treeReduce is the same as treefoldr except that it must have $O(log\ n)$ span assuming g is associative and init is an identity for g |
| findN | ('a → bool) → ('a * 'a → bool) → 'a shrub → int → ('a list → 'b) → (unit → 'b) → 'b | (findN p eq T n s k) returns s [x1, …, xn] where [x1, …, xn] are the leftmost values for T such that for all $i$ from 1 to $n$, p xi returns true and the xi's are eq-distinct. (findN p eq T n s k) returns k() if no such [x1, …, xn] exist |
| sat | prop → ((string * bool) list → 'a) → (unit → 'a) → 'a | sat takes in a proposition, a success function s from a list assigning booleans to free variables to 'a, and a failure function from unit to 'a. If the proposition is satisfiable by an assignment of free variables A, then sat returns s(A). Otherwise, it returns k() |
| findPartition | 'a list → ('a list → bool) → ('a list → bool) → bool | (findParition A pL pR) returns true if there exist an L and R such that (L, R) is a partition of A where pL accepts L and pR accepts R. (findPartition A pL pR) returns false otherwise |

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
datatype 'a shrub = Leaf of 'a | Branch of 'a shrub * 'a shrub
datatype prop = Const of bool | Var of string | Not of prop
              | And of prop * prop | Or of prop * prop
```

Fig. 18. Datatype declarations assumed by homework assignments

respectively), the number of equivalent classes containing more than 1 submission (Non-singleton ECs), and the percentage of submissions found equivalent to at least one other submissions (% in

Table 2. Analysis of the number of equivalent classes (ECs)

| | # | ECs | 90th Percentile ECs | 75th Percentile ECs | Non-singleton ECs | % in Non-singleton ECs |
|---|---|---|---|---|---|---|
| treefoldr | 332 | 22 | 3 | 1 | 9 | 96 |
| integrate | 323 | 34 | 4 | 1 | 5 | 91 |
| slowDoop | 347 | 30 | 6 | 2 | 12 | 95 |
| countNonZero | 351 | 29 | 8 | 4 | 13 | 95 |
| concat | 351 | 40 | 8 | 2 | 10 | 91 |
| treeReduce | 332 | 57 | 24 | 8 | 20 | 89 |
| prefixSum | 351 | 68 | 33 | 7 | 23 | 87 |
| differentiate | 316 | 65 | 34 | 3 | 6 | 81 |
| quicksort | 347 | 73 | 39 | 9 | 18 | 84 |
| findN | 330 | 73 | 40 | 7 | 18 | 83 |
| findPartition | 331 | 83 | 50 | 12 | 22 | 82 |
| sat | 318 | 104 | 73 | 25 | 14 | 72 |

Table 3. Analysis of correctness of student submissions

| | Correct Submissions | Correct ECs | Non-singleton Correct ECs | Incorrect Submissions | Incorrect ECs | Non-singleton Incorrect ECs |
|---|---|---|---|---|---|---|
| treefoldr | 302 | 12 | 4 | 30 | 10 | 5 |
| integrate | 307 | 23 | 3 | 16 | 11 | 2 |
| slowDoop | 346 | 29 | 12 | 1 | 1 | 0 |
| countNonZero | 346 | 26 | 12 | 5 | 3 | 1 |
| concat | 336 | 30 | 9 | 15 | 10 | 1 |
| treeReduce | 188 | 18 | 8 | 144 | 39 | 12 |
| prefixSum | 347 | 64 | 23 | 4 | 4 | 0 |
| differentiate | 308 | 59 | 5 | 8 | 6 | 1 |
| quicksort | 328 | 56 | 16 | 19 | 17 | 2 |
| findN | 296 | 48 | 14 | 34 | 25 | 4 |
| findPartition | 291 | 59 | 13 | 40 | 24 | 9 |
| sat | 273 | 72 | 11 | 45 | 32 | 3 |

Non-singleton ECs). Table 3 shows the number of correct and incorrect student submissions, the number of equivalence classes containing only correct or incorrect submissions, and the number of equivalence classes containing multiple correct or incorrect submissions. There were no equivalence classes that contained both correct and incorrect submissions.

A common trend among all tasks was that a significant majority of student submissions were placed into a relatively small number of large equivalence classes, with the remaining submissions widely dispersed among many small equivalence classes, frequently of size 1. For instance, for the task concat, ZEUS detected 40 equivalent classes. However, only 10 of those classes contain more than one submission, and 8 equivalence classes contain more than 90% of the submissions. In almost all tasks, the largest equivalence classes consisted of various distinct but correct solutions to

Table 4. Runtime analysis

| | # | Total Time (s) | Number of Comparisons | Average Time (s) |
|---|---|---|---|---|
| treefoldr | 332 | 33.701 | 694 | 0.049 |
| integrate | 323 | 45.833 | 991 | 0.046 |
| slowDoop | 347 | 57.564 | 1,201 | 0.048 |
| countNonZero | 351 | 72.268 | 1,578 | 0.046 |
| concat | 351 | 76.110 | 1,614 | 0.047 |
| treeReduce | 332 | 146.830 | 3,089 | 0.048 |
| prefixSum | 351 | 205.695 | 4,112 | 0.050 |
| differentiate | 316 | 140.797 | 3,025 | 0.047 |
| quicksort | 347 | 210.554 | 4,303 | 0.049 |
| findN | 330 | 202.577 | 3,660 | 0.055 |
| findPartition | 331 | 284.502 | 4,807 | 0.059 |
| sat | 318 | 486.218 | 6,532 | 0.074 |

the problem. The one exception to this trend was that in the task treeReduce, a significant number of students mistook associativity for commutativity or otherwise assumed that the function passed into treeReduce was necessarily commutative. This common misunderstanding resulted in a large number of incorrect submissions for treeReduce, but because the misunderstanding was common, ZEUS was still able to place the majority of incorrect submissions into a small number of large equivalence classes. In all tasks, at least 72% of submissions were identified as equivalent to at least one other submission. These results support the hypothesis that ZEUS can be used as a grading assistant to reduce the workload of instructors in reviewing equivalent code, thus freeing their time to provide more detailed feedback.

### 7.4 Runtime performance

Table 4 shows the time needed by ZEUS to cluster all assignments for a given task when running on a common Mac laptop with a 1.6GHz processor and 4 GB of RAM. Specifically, for each task, it shows the number of submissions, the total time to cluster submissions in seconds, the number of pairwise comparisons performed during clustering, and the average time for a single comparison in seconds. The average time to compare two individual submissions is small and it ranges from 0.046 seconds to 0.074 seconds. When performing the clustering of a given assignment, we can observe that the number of comparisons is much less than quadratic and that the total time varies between 1 and 8 minutes. This shows that ZEUS is efficient in practice and can be used in real-time to help instructors grade assignments.

### 7.5 Discussion

We manually inspected the cases where ZEUS did not put two programs in the same equivalence class. The most common reasons for this were the following:

- *The two programs are not equivalent:* since these programs correspond to actual student submissions, not all of the programs are correct. When an incorrect implementation produces the wrong output on any number of inputs, our algorithm appropriately puts it in a different equivalence class from the correct submissions. Additionally, for the sat task, the correct behavior of this function when an input proposition is satisfiable by multiple assignments is not fully defined. If multiple assignments A satisfy the proposition, there are no rules about which A to use when returning s(A). So for this task, two correct submissions could produce different outputs.

- *The two programs use different recursive helper functions:* we found cases where equivalence classes were distinguished by the structure of the helper functions students created. Since our current inference rules do not consider these cases, ZEUS fails to recognize that two programs are equivalent if they use recursive helper functions with different input structures.
- *The two recursive programs use different base cases:* our algorithm's treatment of fixed points causes it to never peer into a recursive call. Our algorithm's treatment of case expressions causes it to only recognize two expressions as equivalent if they handle all inputs in basically the same way. Together, these have the implication that when one expression treats a certain input as a base case while the other expression treats it as a recursive case, then the algorithm will be unable to recognize the expressions as equivalent.
- *One of the programs uses built-in Standard ML functions:* seven out of the twelve tasks involve list manipulation operations. For instance, the top five tasks with the largest number of equivalence classes (sat, findPartition, findN, quicksort, and prefixSum) correspond to tasks that involve list manipulation. Many of the submissions for these tasks use built-in Standard ML functions for list reversal or list concatenation. We did not use a theory of list structures in our SMT Solver, so we were only able to recognize two expressions as equivalent if they used these built-in functions on the same input inputs and order or if they did not use these built-in functions at all.

We note that even with the current limitations, ZEUS already shows that it can efficiently cluster the majority of the submissions into a few equivalence classes. Also, ZEUS could be extended by adding additional inference rules or support for additional SMT theories that would allow the identification of equivalent programs that are currently missed by ZEUS.

## 8 RELATED WORK

Proving that two problems are equivalent is a well-studied topic and has many applications ranging from hardware equivalence [Berman and Trevillyan 1989], compiler optimizations [Zuck et al. 2002], to program equivalence [Godlin and Strichman 2009]. However, the use of program equivalence for grading programming assignments is scarce [Kaleeswaran et al. 2016]. In this section, we cover related work from program equivalence and automatic grading that is closer to our approach.

### 8.1 Program Equivalence

*Program Verification.* The problem of program equivalence can be reduced to a verification problem by showing that both programs satisfy the same specification. For instance, model-checking techniques [Clarke et al. 2004, 2001] can be used to show that two C programs satisfy the same specification. This specification can be written to ensure that for the same input, the programs are equivalent if they always produce the same output. Fedyukovich et al. [Fedyukovich et al. 2016] present techniques for proving that two similar programs have the same property rather than being equivalent. Their approach requires formally verifying one of the programs and using this proof to check the validity of the property in the other program by establishing a coupling between the two programs. A similar approach can also be done for functional programs. For instance, one could write a formal specification of the functionality of a program in Why3ML [Bobot et al. 2015]. We tried this approach by writing a formal specification for programming assignments for the function `concat`, however, the Why3 framework [Bobot et al. 2015] was not able to prove that the program satisfied the specification. In general, proving the program equivalence concerning a specification is a more challenging task than the one we address in this paper since we can take advantage of program structure to prove that they are equivalent.

*Regression Verification.* In regression verification [Felsing et al. 2014; Godlin and Strichman 2009], the goal is to prove that two versions of a program are equivalent. One approach is to transform loops in programs to recursive procedures and to match the recursive calls in both programs and abstract them via uninterpreted functions [Godlin and Strichman 2009]. Other approaches, use invariant inference techniques to prove the equivalence of programs with loops [Felsing et al. 2014]. By using these techniques, one can encode the two versions of the program into Horn clauses and use constraint solvers to automatically find certain kinds of invariants. Alternatively, one can also use symbolic execution and static analysis to generate summaries of program behaviors that capture the modifications between the programs. These summaries can be encoded into logical formulas and their equivalence can be checked using SMT solvers [Backes et al. 2013]. Our approaches also consider that student submissions are similar but they are not different versions of the same program. Even though we do not use any invariant generation techniques, this is orthogonal to our approach and could increase the number of equivalent classes detected for recursive programs.

*Contextual Equivalence.* There is a broad set of work that targets contextual equivalence for functional programs. Approaches based on step-indexed logical relations [Ahmed et al. 2009; Ahmed 2006; Dreyer et al. 2009] or on bisimulations [Hur et al. 2012; Koutavas and Wand 2006; Sumii and Pierce 2005] have been used to prove context equivalence of functional programs with different fragments of ML that often include finite datatypes and integer references. While these approaches are more theoretical and focus on functional programs with state, we do not support state but can handle pattern matching which is crucial for a practical tool to cluster programming assignments of introductory functional courses. The closest approach to ours is the one recently presented by Jaber [Jaber 2020]. Jaber presents techniques for checking the equivalence of OCaml programs with state. His approach focuses in particular on contextual equivalence and developing a framework in which references can be properly accounted for. Our approach neglects references, as we require programs to be purely functional, but includes a more comprehensive treatment of datatypes. We attempted to compare our ZEUS's performance against Jaber's SyTeCi prototype, but unfortunately, all of our benchmarks included datatypes that were not supported by the available prototype.

## 8.2 Automatic grading

*Clustering similar assignments.* To help instructors to grade programming assignments, several automatic techniques have been proposed to cluster similar assignments into buckets with the purpose of giving automatic feedback [Gulwani et al. 2018; Kaleeswaran et al. 2016; Pu et al. 2016; Wang et al. 2018]. Our approach differs from these since our goal is not to replace the instructor or to fully automate the grading but rather to use ZEUS as a grading assistant with formal guarantees.

CLARA [Gulwani et al. 2018] cluster correct programs and selects a canonical program from each cluster to be considered as the reference solution. In this approach, a pair of programs $p_1$ and $p_2$ are said to be dynamic equivalent if they have the same control-flow and if related variables in $p_1$ and $p_2$ always have the same values, in the same order, during the program execution on the same inputs. In contrast, our approach has a stronger notion of equivalence since we do not depend on dynamic program analysis. CodeAssist [Kaleeswaran et al. 2016] clusters submissions for dynamic programming assignments by their solution strategy. They consider a small set of features and if two programs share these features then they are put in the same cluster. Other clustering approaches are based on deep learning techniques [Pu et al. 2016] and also provide no formal guarantees about the quality of the clustering. SemCluster [Perry et al. 2019] improves upon other clustering techniques by considering semantic program features. They use control flow features and data flow features to represent each program and merge this information to create a program feature vector. K-means clustering is used to cluster all programs based on the program feature vectors. Even though there are no formal guarantees for the equivalence of programs in each cluster, experimental results [Perry et al. 2019] show that the number of clusters found by SemCluster is much smaller than competitive approaches.

*Automatic repair.* AutoGrader [Singh et al. 2013] takes as input a reference solution and an error model that consists of potential corrections and uses constraint solving techniques to find a minimum number of corrections that can be used to repair the incorrect student solution. Sarfgen [Wang et al. 2018] uses a three-stage algorithm based on search, align, and repair. It starts by searching for a small number of correct programs that can be used to repair the incorrect submission and have the same control-flow structure. Next, they compute a syntactic distance between those programs using an embedding of ASTs into numerical vectors. These programs are then aligned and the differences between aligned statements can suggest corrections that can be repaired automatically.

Automatic repair is better suited for Massive Open Online Courses where a fully automated method is needed, while our approach is better suited for large, in-person courses, where the feedback of instructors can be more beneficial. The feedback returned by automatic repair tools is limited to changes in the code, while our approach is meant to assist instructors to provide more detailed feedback for students. Each equivalent class will have specific comments that are more helpful to the student than a repaired version of their submission. Moreover, while our approach can be used for both correct and incorrect submissions, automatic repair is only useful to fix incorrect submissions and cannot give any feedback for different implementations of correct submissions.

*Formal guarantees.* Liu et al. [Liu et al. 2019] proposes to automatically determine the correctness of an assignment against a reference solution. Instead of using test cases, they use symbolic execution to search for semantically different execution paths between a student's submission and the reference solution. If such paths exist, then the submission is considered incorrect and feedback can be provided by using counterexamples based on path deviations. Our approach is not based on symbolic execution but instead uses inference rules to derive a formula for which both student submissions are equivalent if and only if they have the same structure and the observable behavior.

CodeAssist [Kaleeswaran et al. 2016] checks equivalence of a candidate submission from a cluster with a correct solution of that cluster that has been previously validated by an instructor. They exploit the fact of just handling dynamic programming assignments to establish a correspondence between variables and control locations of the two programs. Using this correspondence, they can encode the problem into SMT and prove program equivalence. Our approach is more general since our inference rules simulate relationships between expressions of the two programs and can be applied to several problem domains and not just dynamic programming assignments.

## 9 CONCLUSION

We present techniques for checking for equivalence between purely functional programs. Guided by inference rules that inform needed equivalences between two programs' subexpressions, our approach simultaneously deconstructs the expressions being compared to build up a formula that is valid only if the expressions are equivalent. We prove the soundness of our approach: if our approach takes in two expressions of the same type and outputs a valid formula, then the two expressions are equivalent. We implement our approach and show that it can assist grading by clustering over 4,000 real student code submissions from an introductory functional programming class taught at the undergraduate level.

## ACKNOWLEDGMENTS

This work was partially funded by National Science Foundation (Grants CCF-1901381, CCF-1762363, and CCF-1629444).

## REFERENCES


Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proc. Symposium on Principles of Programming Languages*. ACM, 340–353.

Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proc. European Symposium on Programming*. Springer, 69–83.

John D. Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression Verification Using Impact Summaries. In *Proc. International Symposium Model Checking Software*. Springer, 99–116.

C Leonard Berman and Louise H Trevillyan. 1989. Functional comparison of logic designs for VLSI circuits. In *Proc. International Conference on Computer-Aided Design*. IEEE, 456–459.

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2015. Let's verify this with Why3. *Int. J. Softw. Tools Technol. Transf.* 17, 6 (2015), 709–727.

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.

Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods Syst. Des.* 19, 1 (2001), 7–34.

Joshua Clune, Vijay Ramamurthy, Ruben Martins, and Umut A. Acar. 2020. Program Equivalence for Assisted Grading of Functional Programs (Extended Version). *CoRR* (2020).

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *Proc. Annual Symposium on Logic in Computer Science*. IEEE Computer Society, 71–80.

Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2016. Property Directed Equivalence via Abstract Simulation. In *Proc. International Conference Computer-Aided Verification*. Springer, 433–453.

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *Proc. International Conference on Automated Software Engineering*. ACM, 349–360.

Benny Godlin and Ofer Strichman. 2009. Regression verification. In *Proc. Design Automation Conference*. ACM, 466–471.

Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 465–480.

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *Proc. Symposium on Principles of Programming Languages*. ACM, 59–72.

Guilhem Jaber. 2020. SyTeCi: automating contextual equivalence for higher-order programs with references. *PACMPL* 4, POPL (2020), 59:1–59:28.

Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *Proc. International Symposium on Foundations of Software Engineering*. ACM, 739–750.

Vasileios Koutavas and Mitchell Wand. 2006. Small bisimulations for reasoning about higher-order imperative programs. In *Proc. Symposium on Principles of Programming Languages*. ACM, 141–152.

Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. 2019. Automatic grading of programming assignments: an approach based on formal semantics. In *Proc. International Conference on Software Engineering: Software Engineering Education and Training*. IEEE / ACM, 126–137.

David Mitchel Perry, Dohyeong Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: clustering of imperative programming assignments based on quantitative semantic features. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. 860–873.

Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. sk_p: a neural program corrector for MOOCs. In *Proc. International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, 39–40.

Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 15–26.

Eijiro Sumii and Benjamin C. Pierce. 2005. A bisimulation for type abstraction and recursion. In *Proc. Symposium on Principles of Programming Languages*. ACM, 63–74.

Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 481–495.

Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. 2002. VOC: A translation validator for optimizing compilers. *Electronic notes in theoretical computer science* 65, 2 (2002), 2–18.