

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 led 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/>

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

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

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

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

90 The differences between our approach and other state-of-the-art automatic graders and clustering
91 techniques [Gulwani et al. 2018; Perry et al. 2019; Wang et al. 2018] stem from differences in
92 motivation. Since each bucket generated by our approach is meant to be graded by a human, it is
93 more important for our technique to distinguish nonequivalent submissions than to ensure that all
94 equivalent submissions are placed in the same bucket. Ensuring that all equivalent submissions are
95 placed in the same bucket reduces time spent grading equivalent programs, enabling instructors to
96 spend more time giving detailed feedback. This is an important goal, but it is of lower priority than
97 preserving the accuracy of human feedback because it does not matter how detailed feedback is if
98

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.

```

120
121 fun add_opt x y =
122     case (x, y) of
123         (SOME m, SOME n) =>
124             SOME (m + n)
125     | (NONE, _) => NONE
126     | (_, NONE) => NONE
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

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.

```

148 fun split [] = ([], [])
149 | split [x] = ([x], [])
150 | split (x::y::L) =
151   let
152     val (A, B) = split L
153   in
154     (x::A, y::B)
155   end
156
157 fun merge([], L) = L
158 | merge(L, []) = L
159 | merge(x::xs, y::ys) =
160   if x < y
161   then x :: merge (xs, y::ys)
162   else y :: merge (x::xs, ys)
163
164 fun msort [] = []
165 | msort [x] = [x]
166 | msort L =
167   let
168     val (A, B) = split L
169   in
170     merge(msort A, msort B)
171   end
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196

```

```

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

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

197	<i>base types</i>	$b ::= \text{int} \mid \text{boolean}$	
198	<i>types</i>	$\tau ::= b$	<i>base type</i>
199		$\mid \delta$	<i>data type</i>
200		$\mid \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\}$	<i>product type</i>
201		$\mid \tau_1 \rightarrow \tau_2$	<i>function type</i>
202	<i>injection labels</i>	$i ::= \text{label}_1 \mid \text{label}_2 \mid \dots$	
203	<i>patterns</i>	$p ::= _$	<i>wildcard pattern</i>
204		$\mid x$	<i>variable pattern</i>
205		$\mid \{\ell_1 = p_1, \dots, \ell_n = p_n\}$	<i>record pattern</i>
206		$\mid x \text{ as } p$	<i>alias pattern</i>
207		$\mid c$	<i>constant pattern</i>
208		$\mid i \cdot p$	<i>injection pattern (with argument)</i>
209	<i>primitive operations</i>	$o ::= + \mid - \mid * \mid < \mid > \mid \leq \mid \geq$	<i>injection pattern (without argument)</i>
210	<i>expressions</i>	$e ::= c$	<i>constant</i>
211		$\mid x$	<i>variable</i>
212		$\mid \{\ell_1 = e_1, \dots, \ell_n = e_n\}$	<i>record</i>
213		$\mid e \cdot \ell_i$	<i>projection</i>
214		$\mid i \cdot e$	<i>injection (with argument)</i>
215		$\mid i$	<i>injection (without argument)</i>
216		$\mid \text{case } e \{p_1.e_1 \mid \dots \mid p_n.e_n\}$	<i>case analysis</i>
217		$\mid \lambda x.e$	<i>abstraction</i>
218		$\mid e_1 e_2$	<i>application</i>
219		$\mid \text{fix } x \text{ is } e$	<i>fixed point</i>
220		$\mid o$	<i>primitive operation</i>

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 τ to produce an expression of data type δ , and a fixed set of judgments of the form $i : \delta$ for injection labels of data type δ that do not take in an argument. We take $i : \tau \hookrightarrow \delta$ to mean that the type δ has a label i which accepts an argument of type τ , and we take $i : \delta$ to mean that the type δ has a label i that does not accept an argument. Note that by allowing τ to contain instances of δ , this data type system affords LambdaPix 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 τ can be matched against

$$\begin{array}{c}
246 \\
247 \\
248 \\
249 \\
250 \\
251 \\
252 \\
253 \\
254
\end{array}
\frac{}{_ :: \tau \dashv} \text{PATTY}_1 \quad \frac{}{x :: \tau \dashv x : \tau} \text{PATTY}_2 \quad \frac{p_1 :: \tau_1 \dashv \Gamma_1 \quad \dots \quad p_n :: \tau_n \dashv \Gamma_n}{\{l_1 = p_1, \dots, l_n = p_n\} :: \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\} \dashv \Gamma_1 \dots \Gamma_n} \text{PATTY}_3$$

$$\frac{p :: \tau \dashv \Gamma}{x \text{ as } p :: \tau \dashv \Gamma, x : \tau} \text{PATTY}_4 \quad \frac{}{c :: b \dashv} \text{PATTY}_5 \quad \frac{i : \delta}{i :: \delta \dashv} \text{PATTY}_6 \quad \frac{i : \tau \hookrightarrow \delta \quad p :: \tau \dashv \Gamma}{i \cdot p :: \delta \dashv \Gamma} \text{PATTY}_7$$

Fig. 4. Pattern typing in LambdaPix

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

$$\begin{array}{c}
255 \\
256 \\
257 \\
258 \\
259 \\
260 \\
261 \\
262 \\
263 \\
264 \\
265 \\
266 \\
267 \\
268 \\
269 \\
270 \\
271
\end{array}
\frac{}{\Gamma \vdash c : b} \text{TY}_1 \quad \frac{}{\Gamma, x : \tau \vdash x : \tau} \text{TY}_2 \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1, \dots, l_n = e_n\} : \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\}} \text{TY}_3$$

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

$$\frac{\Gamma \vdash e : \tau \quad p_1 :: \tau \dashv \Gamma_1 \quad \Gamma, \Gamma_1 \vdash e_1 : \tau' \quad \dots \quad p_n :: \tau \dashv \Gamma_n \quad \Gamma, \Gamma_n \vdash e_n : \tau'}{\Gamma \vdash \text{case } e \{p_1.e_1 \mid \dots \mid p_n.e_n\} : \tau'} \text{TY}_7 \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2} \text{TY}_8$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \text{TY}_9 \quad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{fix } x \text{ is } e : \tau} \text{TY}_{10} \quad \frac{}{\Gamma, o : \tau_1 \rightarrow \tau_2 \vdash o : \tau_1 \rightarrow \tau_2} \text{TY}_{11}$$

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 τ such that $\Gamma_{\text{initial}} \vdash e : \tau$ is derivable from Figure 5's typing rules, where Γ_{initial} only contains typing judgments for primitive operations of the form $o : \tau_1 \rightarrow \tau_2$.

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 $e \text{ val}$ 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 Γ 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 $e \text{ val}$.

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

To define evaluation we first define two helper judgments to deal with pattern matching (Figure 6). The judgment $v \parallel p \dashv B$ 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 $v \not\parallel p$ means the expression v does not match to the pattern p . It is assumed as a precondition to

$$\begin{array}{c}
 \frac{}{v // _ \dashv} \text{MATCH}_1 \qquad \frac{}{v // x \dashv v/x} \text{MATCH}_2 \\
 \frac{v_1 // p_1 \dashv B_1 \quad \dots \quad v_n // p_n \dashv B_n}{\{\ell_1 = v_1, \dots, \ell_n = v_n\} // \{\ell_1 = p_1, \dots, \ell_n = p_n\} \dashv B_1 \dots B_n} \text{MATCH}_3 \\
 \frac{v_i // p_i}{\{\ell_1 = v_1, \dots, \ell_n = v_n\} // \{\ell_1 = p_1, \dots, \ell_n = p_n\}} \text{MATCH}_4 \qquad \frac{v // p \dashv B}{v // x \text{ as } p \dashv B, v/x} \text{MATCH}_5 \\
 \frac{v // p}{v // x \text{ as } p} \text{MATCH}_6 \qquad \frac{c_1 = c_2}{c_1 // c_2 \dashv} \text{MATCH}_7 \qquad \frac{c_1 \neq c_2}{c_1 // c_2} \text{MATCH}_8 \qquad \frac{}{i // i \dashv} \text{MATCH}_9 \\
 \frac{i_1 \neq i_2}{i_1 // i_2} \text{MATCH}_{10} \qquad \frac{v // p \dashv B}{i \cdot v // i \cdot p \dashv B} \text{MATCH}_{11} \qquad \frac{i_1 \neq i_2}{i_1 \cdot v // i_2 \cdot p} \text{MATCH}_{12} \qquad \frac{v // p}{i \cdot v // i \cdot p} \text{MATCH}_{13} \\
 \frac{}{i_1 \cdot v // i_2} \text{MATCH}_{14} \qquad \frac{}{i_1 // i_2 \cdot p} \text{MATCH}_{15}
 \end{array}$$

Fig. 6. Pattern matching in LambdaPix

these judgements that $v \text{ val}$, $\vdash v : \tau$, 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 $v // p \dashv B$, or $v // p$. It is never simultaneously the case that $v // p \dashv B$ and $v // p$.

$$\begin{array}{c}
 \frac{}{c \text{ val}} \text{DYN}_1 \qquad \frac{e_1 \text{ val} \quad e_2 \text{ val} \quad \dots \quad e_{i-1} \text{ val} \quad e_i \mapsto e'_i}{\{\dots, \ell_i = e_i, \dots\} \mapsto \{\dots, \ell_i = e'_i, \dots\}} \text{DYN}_2 \qquad \frac{e_1 \text{ val} \quad \dots \quad e_n \text{ val}}{\{\ell_1 = e_1, \dots, \ell_n = e_n\} \text{ val}} \text{DYN}_3 \\
 \frac{e \mapsto e'}{e \cdot \ell_j \mapsto e' \cdot \ell_j} \text{DYN}_4 \qquad \frac{\{\dots, \ell_i = e_i, \dots\} \text{ val}}{\{\dots, \ell_i = e_i, \dots\} \cdot \ell_j \mapsto e_j} \text{DYN}_5 \qquad \frac{e \mapsto e'}{i \cdot e \mapsto i \cdot e'} \text{DYN}_6 \qquad \frac{e \text{ val}}{i \cdot e \text{ val}} \text{DYN}_7 \\
 \frac{}{i \text{ val}} \text{DYN}_8 \qquad \frac{e \mapsto e'}{\text{case } e \{p_1.e_1 \mid \dots \mid p_n.e_n\} \mapsto \text{case } e' \{p_1.e_1 \mid \dots \mid p_n.e_n\}} \text{DYN}_9 \\
 \frac{e \text{ val} \quad e // p_1 \quad \dots \quad e // p_{i-1} \quad e // p_i \dashv B}{\text{case } e \{\dots \mid p_i.e_i \mid \dots\} \mapsto [B]e_i} \text{DYN}_{10} \qquad \frac{}{\lambda x.e \text{ val}} \text{DYN}_{11} \qquad \frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \text{DYN}_{12} \\
 \frac{e_1 \text{ val} \quad e_2 \mapsto e'_2}{e_1 e_2 \mapsto e_1 e'_2} \text{DYN}_{13} \qquad \frac{e_2 \text{ val}}{(\lambda x.e) e_2 \mapsto [e_2/x]e} \text{DYN}_{14} \qquad \frac{}{\text{fix } x \text{ is } e \mapsto [\text{fix } x \text{ is } e/x]e} \text{DYN}_{15} \\
 \frac{}{o \text{ val}} \text{DYN}_{16} \qquad \frac{e \text{ val}}{o e \mapsto e'} \text{DYN}_{17} \qquad \frac{v \text{ val}}{v \mapsto v} \text{BIGDYN}_1 \qquad \frac{e \mapsto e' \quad e' \mapsto v}{e \mapsto v} \text{BIGDYN}_2
 \end{array}$$

Fig. 7. Dynamic semantics of LambdaPix

In Figure 7 we use these helper judgments to define the evaluation judgments. In 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 BIGDYN_1 and BIGDYN_2 , big-step dynamics are defined as the transitive closure of the small-step dynamics.

4 SOUND EQUIVALENCE INFERENCE

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

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

Fig. 8. Logic Formulas

$$\begin{array}{cccccc}
 \frac{}{c \text{ Term}} \text{TERM}_1 & \frac{}{x \text{ Term}} \text{TERM}_2 & \frac{t_1 \text{ Term} \quad t_2 \text{ Term} \quad \dots \quad t_n \text{ Term}}{\{\ell_1 = t_1, \dots, \ell_n = t_n\} \text{ Term}} \text{TERM}_3 & \frac{t \text{ Term}}{t \cdot \ell_i \text{ Term}} \text{TERM}_4 \\
 \frac{t \text{ Term}}{i \cdot t \text{ Term}} \text{TERM}_5 & \frac{}{i \text{ Term}} \text{TERM}_6 & \frac{}{_ \text{ Term}} \text{TERM}_7 & \frac{t \text{ Term}}{x \text{ as } t \text{ Term}} \text{TERM}_8 & \frac{}{o \text{ Term}} \text{TERM}_9 \\
 & & \frac{t \text{ Term}}{o \text{ } t \text{ Term}} \text{TERM}_{10} & & &
 \end{array}$$

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: $x \text{ as } e_1 \equiv e_2$, 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. $[3/x](x \equiv 1 \wedge x \equiv 2)$ yields $3 \equiv 1 \wedge 3 \equiv 2$). 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 ($\forall_{\Gamma}^{\text{val}}.j$). If $\Gamma = \vec{x} : \vec{\tau}$, then the judgement $\forall_{\Gamma}^{\text{val}}.j$ 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 Γ_{initial} , and therefore Γ , we omit typings of the form $o : \tau_1 \rightarrow \tau_2$ from $\vec{x} : \vec{\tau}$ so that we do not range over all possible meanings for LambdaPix's primitive operations. Then if Γ is a typing context with a mapping for every free variable in a formula σ , the validity of σ is denoted $\forall_{\Gamma}^{\text{val}}.\sigma$.

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.

$$\begin{array}{c}
 \frac{e \rightsquigarrow e' \quad e' \downarrow e''}{e \downarrow e''} \text{BIGWHNF}_1 \qquad \frac{e \not\rightsquigarrow}{e \downarrow e} \text{BIGWHNF}_2 \qquad \frac{e_1 \rightsquigarrow e'_1}{e_1 e_2 \rightsquigarrow e'_1 e_2} \text{WHNF}_1 \\
 \\
 \frac{}{(\lambda x.e_1)e_2 \rightsquigarrow [e_2/x]e_1} \text{WHNF}_2 \qquad \frac{e \rightsquigarrow e'}{e \cdot \ell_i \rightsquigarrow e' \cdot \ell_i} \text{WHNF}_3 \qquad \frac{}{\{\dots, \ell_i = e, \dots\} \cdot \ell_i \rightsquigarrow e} \text{WHNF}_4
 \end{array}$$

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 LambdaPix, 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 $\text{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p'.e'_1, p'.e'_2)$ defined in Figure 12 does exactly that, taking in two bindings and returning freshened versions of those bindings

$$\begin{array}{c}
442 \\
443 \\
444 \\
445 \\
446 \\
447 \\
448 \\
449 \\
450 \\
451 \\
452 \\
453 \\
454 \\
455 \\
456 \\
457 \\
458 \\
459 \\
460 \\
461 \\
462 \\
463 \\
464 \\
465 \\
466 \\
467 \\
468 \\
469 \\
470 \\
471 \\
472 \\
473 \\
474 \\
475 \\
476 \\
477 \\
478 \\
479 \\
480 \\
481 \\
482 \\
483 \\
484 \\
485 \\
486 \\
487 \\
488 \\
489 \\
490
\end{array}$$

$$\begin{array}{c}
\frac{}{\text{freshen } _e \hookrightarrow _e} \text{FRESHEN}_1 \qquad \frac{y \text{ fresh}}{\text{freshen } x.e \hookrightarrow y.[y/x]e} \text{FRESHEN}_2 \\
\frac{\text{freshen } p_1.e \hookrightarrow p'_1.e_1 \quad \text{freshen } p_2.e_1 \hookrightarrow p'_2.e_2 \quad \dots \quad \text{freshen } p_n.e_{n-1} \hookrightarrow p'_n.e_n}{\text{freshen } \{\ell_1 = p_1, \dots, \ell_n = p_n\}.e \hookrightarrow \{\ell_1 = p'_1, \dots, \ell_n = p'_n\}.e_n} \text{FRESHEN}_3 \\
\frac{y \text{ fresh} \quad \text{freshen } p.e \hookrightarrow p'.e'}{\text{freshen } x \text{ as } p.e \hookrightarrow y \text{ as } p'.[y/x]e'} \text{FRESHEN}_4 \qquad \frac{}{\text{freshen } c.e \hookrightarrow c.e} \text{FRESHEN}_5 \\
\frac{}{\text{freshen } i.e \hookrightarrow i.e} \text{FRESHEN}_6 \qquad \frac{\text{freshen } p.e \hookrightarrow p'.e'}{\text{freshen } i \cdot p.e \hookrightarrow i \cdot p'.e'} \text{FRESHEN}_7
\end{array}$$

Fig. 11. Freshening

$$\begin{array}{c}
457 \\
458 \\
459 \\
460 \\
461 \\
462 \\
463 \\
464 \\
465 \\
466 \\
467 \\
468 \\
469 \\
470 \\
471 \\
472 \\
473 \\
474 \\
475 \\
476 \\
477 \\
478 \\
479 \\
480 \\
481 \\
482 \\
483 \\
484 \\
485 \\
486 \\
487 \\
488 \\
489 \\
490
\end{array}$$

$$\begin{array}{c}
\frac{}{\text{EB}(_e_1, _e_2) \hookrightarrow (_e_1, _e_2)} \text{EB}_1 \qquad \frac{y \text{ fresh}}{\text{EB}(_e_1, x.e_2) \hookrightarrow (y.e_1, y.[y/x]e_2)} \text{EB}_2 \\
\frac{y \text{ fresh}}{\text{EB}(x.e_1, _e_2) \hookrightarrow (y.[y/x]e_1, y.e_2)} \text{EB}_3 \qquad \frac{y \text{ fresh}}{\text{EB}(x.e_1, x'.e_2) \hookrightarrow (y.[y/x]e_1, y.[y/x']e_2)} \text{EB}_4 \\
\frac{y \text{ fresh} \quad \text{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p'.e'_1, p'.e'_2)}{\text{EB}(x \text{ as } p_1.e_1, p_2.e_2) \hookrightarrow (y \text{ as } p'.[y/x]e'_1, y \text{ as } p'.e'_2)} \text{EB}_5 \\
\frac{y \text{ fresh} \quad \text{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p'.e'_1, p'.e'_2)}{\text{EB}(p_1.e_1, x \text{ as } p_2.e_2) \hookrightarrow (y \text{ as } p'.e'_1, y \text{ as } p'.[y/x]e'_2)} \text{EB}_6 \\
\frac{\text{EB}(p_1.e_1, p'_1.e_2) \hookrightarrow (p''_1.e'_1, p''_1.e'_2) \quad \dots \quad \text{EB}(p_n.e_{n-1}, p'_n.e_n) \hookrightarrow (p''_n.e'_{n-1}, p''_n.e'_n)}{\text{EB}(\{\ell_1 = p_1, \dots, \ell_n = p_n\}.e_1, \{\ell_1 = p'_1, \dots, \ell_n = p'_n\}.e_2) \hookrightarrow (\{\ell_1 = p''_1, \dots, \ell_n = p''_n\}.e'_1, \{\ell_1 = p''_1, \dots, \ell_n = p''_n\}.e'_2)} \text{EB}_7 \\
\frac{}{\text{EB}(c.e_1, c.e_2) \hookrightarrow (c.e_1, c.e_2)} \text{EB}_8 \qquad \frac{}{\text{EB}(i.e_1, i.e_2) \hookrightarrow (i.e_1, i.e_2)} \text{EB}_9 \\
\frac{\text{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p'.e'_1, p'.e'_2)}{\text{EB}(i \cdot p_1.e_1, i \cdot p_2.e_2) \hookrightarrow (i \cdot p'.e'_1, i \cdot p'.e'_2)} \text{EB}_{10}
\end{array}$$

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 $\text{FT}(\{p_1.e_1 \mid \dots \mid p_n.e_n\}, \{p'_1.e'_1 \mid \dots \mid p'_m.e'_m\}) \stackrel{s}{\hookrightarrow} (\{p''_1.e''_1 \mid \dots \mid p''_n.e''_n\}, \{p'''_1.e'''_1 \mid \dots \mid p'''_m.e'''_m\})$ 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

$$\begin{array}{c}
\text{491} \\
\text{492} \\
\text{493} \\
\text{494} \\
\text{495} \\
\text{496} \\
\text{497} \\
\text{498} \\
\text{499} \\
\text{500} \\
\text{501} \\
\text{502} \\
\text{503} \\
\text{504} \\
\text{505} \\
\text{506} \\
\text{507} \\
\text{508} \\
\text{509} \\
\text{510} \\
\text{511} \\
\text{512} \\
\text{513} \\
\text{514} \\
\text{515} \\
\text{516} \\
\text{517} \\
\text{518} \\
\text{519} \\
\text{520} \\
\text{521} \\
\text{522} \\
\text{523} \\
\text{524} \\
\text{525} \\
\text{526} \\
\text{527} \\
\text{528} \\
\text{529} \\
\text{530} \\
\text{531} \\
\text{532} \\
\text{533} \\
\text{534} \\
\text{535} \\
\text{536} \\
\text{537} \\
\text{538} \\
\text{539}
\end{array}$$

$$\begin{array}{c}
\frac{\text{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p.e'_1, p.e'_2)}{\text{FT}(\{p_1.e_1 \mid \cdot\}, \{p_2.e_2 \mid \cdot\}) \overset{1}{\hookrightarrow} (\{p.e'_1 \mid \cdot\}, \{p.e'_2 \mid \cdot\})} \text{FT}_1 \\
\\
\frac{\text{EB}(p_1.e_1, p_2.e_2) \hookrightarrow (p.e'_1, p.e'_2) \quad \text{FT}(rest_1, rest_2) \overset{n}{\hookrightarrow} (rest'_1, rest'_2)}{\text{FT}(\{p_1.e_1 \mid rest_1\}, \{p_2.e_2 \mid rest_2\}) \overset{n+1}{\hookrightarrow} (\{p.e'_1 \mid rest'_1\}, \{p.e'_2 \mid rest'_2\})} \text{FT}_2 \\
\\
\frac{\forall_{i \in [n]} (\text{freshen } p_i.e_i \hookrightarrow p'_i.e'_i) \quad \forall_{i \in [m]} (\text{freshen } p'_i.e'_i \hookrightarrow p''_i.e''_i)}{\text{FT}(\{p_1.e_1 \mid \dots \mid p_n.e_n\}, \{p'_1.e'_1 \mid \dots \mid p'_m.e'_m\}) \overset{0}{\hookrightarrow} (\{p''_1.e''_1 \mid \dots \mid p''_n.e''_n\}, \{p''_1.e''_1 \mid \dots \mid p''_m.e''_m\})} \text{FT}_3
\end{array}$$

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 FT_2 or FT_3 , it will apply FT_2).

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

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

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

$$\frac{e_1 \downarrow e'_1 \quad e_2 \downarrow e'_2 \quad \Gamma \vdash e'_1 \overset{\sigma}{\hookrightarrow} e'_2 : \tau \dashv \Gamma'}{\Gamma \vdash e_1 \overset{\sigma}{\hookrightarrow} e_2 : \tau \dashv \Gamma'} \text{ IsoExp}$$

Fig. 14. IsoExp Rule

The judgment $\Gamma \vdash e_1 \overset{\sigma}{\hookrightarrow} e_2 : \tau \dashv \Gamma'$ is defined by Figure 14 and is mutually recursive with $\Gamma \vdash e_1 \overset{\sigma}{\hookrightarrow} e_2 : \tau \dashv \Gamma'$. 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 $\Gamma_{\text{initial}} \vdash e_1 \overset{\sigma}{\hookrightarrow} e_2 : \tau \dashv \Gamma'$ and $\bigvee_{\Gamma'} \sigma$.

The purpose of the distinction between the two judgments is to allow our approach to perform weak head reduction exactly when needed. The judgment $\Gamma \vdash e_1 \overset{\sigma}{\hookrightarrow} e_2 : \tau \dashv \Gamma'$ 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, $\text{ISO}_{\text{lambda}}$ has the premises x fresh and $\Gamma, x : \tau \vdash [x/x_1]e_1 \overset{\sigma}{\hookrightarrow} [x/x_2]e_2 : \tau' \dashv \Gamma'$. The former premise simply declares x as a previously unused

$$\begin{array}{c}
540 \quad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad e_1 \text{ Term} \quad e_2 \text{ Term}}{\Gamma \vdash e_1 \xleftrightarrow{e_1 \equiv e_2} e_2 : \tau \dashv} \text{ISO}_{\text{atomic}} \\
541 \\
542 \\
543 \quad \frac{\Gamma \vdash e_1 \xleftrightarrow{\sigma_1} e'_1 : \tau_1 \dashv \Gamma'_1 \quad \dots \quad \Gamma \vdash e_n \xleftrightarrow{\sigma_n} e'_n : \tau_n \dashv \Gamma'_n}{\Gamma \vdash \{\ell_1 = e_1, \dots, \ell_n = e_n\} \xleftrightarrow{\sigma_1 \wedge \dots \wedge \sigma_n} \{\ell_1 = e'_1, \dots, \ell_n = e'_n\} : \{\ell_1 : \tau_1, \dots, \ell_n : \tau_n\} \dashv \Gamma'_1, \dots, \Gamma'_n} \text{ISO}_{\text{record}} \\
544 \\
545 \\
546 \quad \frac{\Gamma \vdash e_1 \xleftrightarrow{\sigma} e_2 : \{\dots, \ell_i : \tau_i, \dots\} \dashv \Gamma'}{\Gamma \vdash e_1 \cdot \ell_i \xleftrightarrow{\sigma} e_2 \cdot \ell_i : \tau_i \dashv \Gamma'} \text{ISO}_{\text{projection}} \quad \frac{i : \tau \hookrightarrow \delta \quad \Gamma \vdash e_1 \xleftrightarrow{\sigma} e_2 : \tau \dashv \Gamma'}{\Gamma \vdash i \cdot e_1 \xleftrightarrow{\sigma} i \cdot e_2 : \delta \dashv \Gamma'} \text{ISO}_{\text{injection}} \\
547 \\
548 \\
549 \\
550 \quad \frac{x \text{ fresh} \quad \Gamma, x : \tau \vdash [x/x_1]e_1 \xleftrightarrow{\sigma} [x/x_2]e_2 : \tau' \dashv \Gamma'}{\Gamma \vdash \lambda x_1. e_1 \xleftrightarrow{\sigma} \lambda x_2. e_2 : \tau \rightarrow \tau' \dashv x : \tau, \Gamma'} \text{ISO}_{\text{lambda}} \\
551 \\
552 \\
553 \quad \frac{x \text{ fresh} \quad \Gamma, x : \tau \vdash [x/x_1]e_1 \xleftrightarrow{\sigma} [x/x_2]e_2 : \tau \dashv \Gamma'}{\Gamma \vdash \text{fix } x_1 \text{ is } e_1 \xleftrightarrow{\sigma} \text{fix } x_2 \text{ is } e_2 : \tau \dashv x : \tau, \Gamma'} \text{ISO}_{\text{fix}} \\
554 \\
555 \\
556 \\
557 \\
558 \\
559 \\
560 \\
561 \\
562 \\
563 \\
564 \\
565 \\
566 \\
567 \\
568 \\
569 \\
570 \\
571 \\
572 \\
573 \\
574 \\
575 \\
576 \\
577 \\
578 \\
579 \\
580 \\
581 \\
582 \\
583 \\
584 \\
585 \\
586 \\
587 \\
588
\end{array}$$

Fig. 15. Formula Generation Rules

variable, and the latter premise states that if any value x of type τ (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 σ 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, $\text{ISO}_{\text{projection}}$ has the premise $\Gamma \vdash e_1 \xleftrightarrow{\sigma} e_2 : \{\dots, \ell_i : \tau_i, \dots\} \dashv \Gamma'$. If this premise holds, then the conclusion that $\Gamma \vdash e_1 \cdot \ell_i \xleftrightarrow{\sigma} e_2 \cdot \ell_i : \tau_i \dashv \Gamma'$ 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, $\text{ISO}_{\text{application1}}$ 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 $\text{ISO}_{\text{application1}}$ 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, $\text{ISO}_{\text{application1}}$ is sufficient to find equivalence because f can be found equivalent to f and x can be found equivalent to $x + 0$ via $\text{ISO}_{\text{atomic}}$. But if $f(x)$ and $f(x) + 0$ are being compared, $\text{ISO}_{\text{application1}}$ 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, $\text{ISO}_{\text{application2}}$ 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 $\text{ISO}_{\text{atomic}}$.

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

$$\begin{array}{c}
589 \\
590 \\
591 \\
592 \\
593 \\
594 \\
595 \\
596 \\
597 \\
598 \\
599 \\
600 \\
601 \\
602 \\
603 \\
604 \\
605 \\
606 \\
607 \\
608 \\
609 \\
610 \\
611 \\
612 \\
613 \\
614 \\
615 \\
616 \\
617 \\
618 \\
619 \\
620 \\
621 \\
622 \\
623 \\
624 \\
625 \\
626 \\
627 \\
628 \\
629 \\
630 \\
631 \\
632 \\
633 \\
634 \\
635 \\
636 \\
637
\end{array}$$

$$\begin{array}{c}
\frac{\Gamma \vdash e_1 \xleftrightarrow{\sigma} e_2 : \tau \rightarrow \tau' \dashv \Gamma' \quad \Gamma \vdash e'_1 \xleftrightarrow{\sigma'} e'_2 : \tau \dashv \Gamma''}{\Gamma \vdash e_1 e'_1 \xleftrightarrow{\sigma \wedge \sigma'} e_2 e'_2 : \tau' \dashv \Gamma', \Gamma''} \text{ISO}_{\text{application1}} \\
\frac{y \text{ fresh } \quad \Gamma, y : \tau \vdash y \xleftrightarrow{\sigma} [y/(x e_1)]e_2 : \tau \dashv \Gamma'}{\Gamma \vdash x e_1 \xleftrightarrow{\sigma} e_2 : \tau \dashv \Gamma'} \text{ISO}_{\text{application2}} \\
\frac{y \text{ fresh } \quad \Gamma, y : \tau \vdash [y/(x e_2)]e_1 \xleftrightarrow{\sigma} y : \tau \dashv \Gamma'}{\Gamma \vdash e_1 \xleftrightarrow{\sigma} x e_2 : \tau \dashv \Gamma'} \text{ISO}_{\text{application3}} \\
\frac{y \text{ fresh } \quad \Gamma, y : \tau \vdash y \xleftrightarrow{\sigma} [y/(o e_1)]e_2 : \tau \dashv \Gamma'}{\Gamma \vdash o e_1 \xleftrightarrow{\sigma} e_2 : \tau \dashv \Gamma'} \text{ISO}_{\text{application4}} \\
\frac{y \text{ fresh } \quad \Gamma, y : \tau \vdash [y/(o e_2)]e_1 \xleftrightarrow{\sigma} y : \tau \dashv \Gamma'}{\Gamma \vdash e_1 \xleftrightarrow{\sigma} o e_2 : \tau \dashv \Gamma'} \text{ISO}_{\text{application5}} \\
\frac{y \text{ fresh } \quad \Gamma, y : \tau \vdash y \xleftrightarrow{\sigma} [y/((\text{fix } x_1 \text{ is } e_1) e_2)]e : \tau \dashv \Gamma'}{\Gamma \vdash (\text{fix } x_1 \text{ is } e_1) e_2 \xleftrightarrow{\sigma} e : \tau \dashv \Gamma'} \text{ISO}_{\text{application6}} \\
\frac{y \text{ fresh } \quad \Gamma, y : \tau \vdash [y/((\text{fix } x_1 \text{ is } e_1) e_2)]e \xleftrightarrow{\sigma} y : \tau \dashv \Gamma'}{\Gamma \vdash e \xleftrightarrow{\sigma} (\text{fix } x_1 \text{ is } e_1) e_2 : \tau \dashv \Gamma'} \text{ISO}_{\text{application7}}
\end{array}$$

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 ISO_{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 case $e \{p_1.e_1 \mid \dots \mid p_n.e_n\}$ 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 σ 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

$$\begin{array}{c}
638 \\
639 \\
640 \\
641 \\
642 \\
643 \\
644 \\
645 \\
646 \\
647 \\
648 \\
649 \\
650 \\
651 \\
652 \\
653 \\
654 \\
655 \\
656 \\
657 \\
658 \\
659 \\
660 \\
661 \\
662 \\
663 \\
664 \\
665 \\
666 \\
667 \\
668 \\
669 \\
670 \\
671 \\
672 \\
673 \\
674 \\
675 \\
676 \\
677 \\
678 \\
679 \\
680 \\
681 \\
682 \\
683 \\
684 \\
685 \\
686
\end{array}$$

$$\begin{array}{c}
\frac{e \text{ Term} \quad \forall_{i \in [n]} \left(\text{freshen } p_i.e_i \hookrightarrow p'_i.e'_i \quad p'_i :: \tau' \dashv \Gamma_i \quad \Gamma, \Gamma_i \vdash e'_i \xleftrightarrow{\sigma_i} e' : \tau \dashv \Gamma'_i \right)}{\Gamma \vdash \text{case } e \{p_1.e_1 \mid \dots \mid p_n.e_n\} \xleftrightarrow{\wedge_{i \in [n]} ((\wedge_{j \in [i-1]} (e \neq p'_j)) \wedge e \equiv p'_i) \Rightarrow \sigma_i} e' : \tau \dashv \forall_{i \in [n]} \Gamma_i, \Gamma'_i} \text{ISO}_{\text{case1}} \\
\frac{e \text{ Term} \quad \forall_{i \in [n]} \left(\text{freshen } p_i.e_i \hookrightarrow p'_i.e'_i \quad p'_i :: \tau' \dashv \Gamma_i \quad \Gamma, \Gamma_i \vdash e'_i \xleftrightarrow{\sigma_i} e' : \tau \dashv \Gamma'_i \right)}{\Gamma \vdash e' \xleftrightarrow{\wedge_{i \in [n]} ((\wedge_{j \in [i-1]} (e \neq p'_j)) \wedge e \equiv p'_i) \Rightarrow \sigma_i} \text{case } e \{p_1.e_1 \mid \dots \mid p_n.e_n\} : \tau \dashv \forall_{i \in [n]} \Gamma_i, \Gamma'_i} \text{ISO}_{\text{case2}} \\
\frac{\Gamma \vdash e \xleftrightarrow{\sigma} e' : \tau' \dashv \Gamma' \quad x \text{ fresh} \quad \Gamma, x : \tau' \vdash \text{case } x \{ \dots \} \xleftrightarrow{\sigma'} \text{case } x \{ \dots' \} : \tau \dashv \Gamma''}{\Gamma \vdash \text{case } e \{ \dots \} \xleftrightarrow{\sigma \wedge \sigma'} \text{case } e' \{ \dots' \} : \tau \dashv \Gamma', x : \tau', \Gamma''} \text{ISO}_{\text{case3}} \\
\frac{\text{FT}(\{M\}, \{M'\}) \xleftrightarrow{\S} (\{p_1.e_1 \mid \dots \mid p_n.e_n\}, \{p'_1.e'_1 \mid \dots \mid p'_m.e'_m\}) \quad \forall_{i \in [s]} (p_i :: \tau' \dashv \Gamma_i \quad \Gamma, \Gamma_i \vdash e_i \xleftrightarrow{\sigma_i} e'_i \dashv \Gamma'_i) \quad \forall_{j \in [s+1, n]} (p_j :: \tau' \dashv \Gamma_j \quad \Gamma, \Gamma_j \vdash e_j \xleftrightarrow{\sigma_j} \text{case } x \{p'_1.e'_1 \mid \dots \mid p'_m.e'_m\} : \tau \dashv \Gamma'_j)}{\Gamma \vdash \text{case } x \{M\} \xleftrightarrow{\Psi} \text{case } x \{M'\} : \tau \dashv \forall_{i \in [n]} \Gamma_i, \Gamma'_i} \text{ISO}_{\text{case4}} \\
\frac{\text{FT}(\{M'\}, \{M\}) \xleftrightarrow{\S} (\{p'_1.e'_1 \mid \dots \mid p'_m.e'_m\}, \{p_1.e_1 \mid \dots \mid p_n.e_n\}) \quad \forall_{i \in [s]} (p_i :: \tau' \dashv \Gamma_i \quad \Gamma, \Gamma_i \vdash e_i \xleftrightarrow{\sigma_i} e'_i \dashv \Gamma'_i) \quad \forall_{j \in [s+1, n]} (p_j :: \tau' \dashv \Gamma_j \quad \Gamma, \Gamma_j \vdash \text{case } x \{p'_1.e'_1 \mid \dots \mid p'_m.e'_m\} \xleftrightarrow{\sigma_j} e_j : \tau \dashv \Gamma'_j)}{\Gamma \vdash \text{case } x \{M'\} \xleftrightarrow{\Psi} \text{case } x \{M\} : \tau \dashv \forall_{i \in [n]} \Gamma_i, \Gamma'_i} \text{ISO}_{\text{case5}} \\
\Psi := (\wedge_{i \in [s]} \sigma_i) \wedge (\wedge_{j \in [s+1, n]} ((\wedge_{k \in [j-1]} (x \neq p_k)) \wedge x \equiv p_j) \Rightarrow \sigma_j)
\end{array}$$

Fig. 17. Formula Generation Case Rules

to generate σ' . This approach is needed in addition to $\text{ISO}_{\text{case1}}$ and $\text{ISO}_{\text{case2}}$ because $\text{ISO}_{\text{case1}}$ and $\text{ISO}_{\text{case2}}$ 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 $\text{ISO}_{\text{case1}}$, $\text{ISO}_{\text{case2}}$, and $\text{ISO}_{\text{case3}}$ 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 $\text{ISO}_{\text{case3}}$ 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 $\sigma = \text{False}$, 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 $\text{ISO}_{\text{projection}}$ 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 ISO_{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.
- $\text{ISO}_{\text{application6}}$ and $\text{ISO}_{\text{application7}}$ 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 $\text{ISO}_{\text{case1}}$ and $\text{ISO}_{\text{case2}}$ 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 = e1 in e2 end` becomes `case e1 {x.e2}` 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.

```

736
737 fun add_opt x y =
738   case (x, y) of
739     (SOME m, SOME n) =>
740       SOME (m + n)
741   | (NONE, _) => NONE
742   | (_, NONE) => NONE
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784

```

```

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 }

(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 $\Gamma \vdash e_1 \xrightarrow{\sigma} e_2 : \tau \dashv \Gamma'$ judgement, defined by the rule [ISOExp](#). By this rule, we reduce both expressions to weak head normal form then apply the $\Gamma \vdash e_1 \xrightarrow{\sigma} e_2 : \tau \dashv \Gamma'$ 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.

```

885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311
3312
3313
3314
3315
3316
3317
33
```

branch we will generate a formula of the form

$$((x, y) \equiv (\text{SOME}\cdot m1, \text{SOME}\cdot n1)) \Rightarrow \dots$$

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 $\text{Iso}_{\text{case2}}$ 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 $\text{Iso}_{\text{atomic}}$ to compare base terms.

Putting everything together, the final formula is:

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

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

$$\begin{aligned} & ((x, y) \equiv (\text{SOME}\cdot m1, \text{SOME}\cdot n1)) \Rightarrow \\ & ((x \equiv \text{SOME}\cdot b1) \Rightarrow \\ & \quad (y \equiv \text{SOME}\cdot b2) \Rightarrow (\text{SOME}\cdot (m1+n1) \equiv \text{SOME}\cdot (b1+b2)) \wedge \\ & \quad (y \neq \text{SOME}\cdot b2 \wedge y \equiv \text{NONE}) \Rightarrow (\text{SOME}\cdot (m1+n1) \equiv \text{NONE}) \\ &) \wedge \\ & ((x \neq \text{SOME}\cdot b1 \wedge x \equiv \text{NONE}) \Rightarrow \\ & \quad (y \equiv \text{SOME}\cdot b2) \Rightarrow (\text{SOME}\cdot (m1+n1) \equiv \text{NONE}) \wedge \\ & \quad (y \neq \text{SOME}\cdot b2 \wedge y \equiv \text{NONE}) \Rightarrow (\text{SOME}\cdot (m1+n1) \equiv \text{NONE}) \\ &) \end{aligned}$$

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 τ .

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

- (1) Rule EQ₁: In the case that $\tau = \tau_1 \rightarrow \tau_2$, for all expressions v such that $\Gamma_{\text{initial}} \vdash v : \tau_1$, $v_1 v \cong v_2 v : \tau_2$.
- (2) Rule EQ₂: In the case that τ is not an arrow type, for all patterns p such that $p :: \tau$, either $v_1 \parallel p \dashv B$ and $v_2 \parallel p \dashv B$ or $v_1 \not\parallel p$ and $v_2 \not\parallel p$.

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

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

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

893
 894 **THEOREM 6.2 (SOUNDNESS).** *For any expressions e_1 and e_2 , if $\Gamma_{\text{initial}} \vdash e_1 \stackrel{\sigma}{\Leftrightarrow} e_2 : \tau \dashv \Gamma'$ and $\forall_{\Gamma'.\sigma}^{\text{val}}$,
 895 then $e_1 \cong e_2 : \tau$.*

897 6.2 Proof Sketch

898 As the $\Gamma \vdash e_1 \stackrel{\sigma}{\Leftrightarrow} e_2 : \tau \dashv \Gamma'$ judgement is defined simultaneously with the $\Gamma \vdash e_1 \stackrel{\sigma}{\Leftrightarrow} e_2 : \tau \dashv \Gamma'$
 899 judgement, we prove the theorem by simultaneous induction on both of these judgements. We also

900 use the $\forall_{\Gamma'.j}^{\text{val}}$ judgement to strengthen the inductive hypotheses to account for variables. Recall
 901 that if $\Gamma = \vec{x} : \vec{\tau}$, then the judgement $\forall_{\Gamma'.j}^{\text{val}}$ holds if for all \vec{v} where $v_i : \tau_i$ and $v_i \text{ val}$ for all $v_i \in \vec{v}$, it is
 902 the case that $[\vec{v}/\vec{x}]j$ holds (implicitly, we omit any primitive operations from the context $\Gamma = \vec{x} : \vec{\tau}$
 903 as to not range over all possible meanings for LambdaPix's primitive operations). The theorem we
 904 wish to show by induction is then:

- 905 • If $\Gamma \vdash e_1 \stackrel{\sigma}{\Leftrightarrow} e_2 : \tau \dashv \Gamma'$ then $\forall_{\Gamma'.j}^{\text{val}} \left(\text{if } \left(\forall_{\Gamma'.\sigma}^{\text{val}} \right) \text{ then } e_1 \cong e_2 : \tau \right)$.
- 906 • If $\Gamma \vdash e_1 \stackrel{\sigma}{\Leftrightarrow} e_2 : \tau \dashv \Gamma'$ then $\forall_{\Gamma'.j}^{\text{val}} \left(\text{if } \left(\forall_{\Gamma'.\sigma}^{\text{val}} \right) \text{ then } e_1 \cong e_2 : \tau \right)$.

907 We first verify that the above statements imply the soundness theorem. Indeed, when $\Gamma_{\text{initial}} \vdash$
 908 $e_1 \stackrel{\sigma}{\Leftrightarrow} e_2 : \tau \dashv \Gamma'$ we have $\forall_{\Gamma_{\text{initial}}.j}^{\text{val}} \left(\text{if } \left(\forall_{\Gamma'.\sigma}^{\text{val}} \right) \text{ then } e_1 \cong e_2 : \tau \right)$. Since Γ_{initial} contains only primitive

909 operations, which are omitted from the $\forall_{\Gamma'.j}^{\text{val}}$ judgment, the outer quantifier quantifies over no
 910 variables, so we have that if $\left(\forall_{\Gamma'.\sigma}^{\text{val}} \right)$ then $e_1 \cong e_2 : \tau$. This together with the assumption that $\forall_{\Gamma'.\sigma}^{\text{val}}$
 911 allows us to conclude that $e_1 \cong e_2 : \tau$.

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

914 **ISO_{record}** : Let $\Gamma = \vec{x} : \vec{\tau}$ and let \vec{v} be arbitrary where $v_i : \tau_i$ and $v_i \text{ val}$ for all $v_i \in \vec{v}$. Assume
 915 $[\vec{v}/\vec{x}] \left(\forall_{\Gamma'_1, \dots, \Gamma'_n}^{\text{val}} \sigma_1 \wedge \dots \wedge \sigma_n \right)$. It must be shown that $[\vec{v}/\vec{x}] (\{ \ell_1 = e_1, \dots, \ell_n = e_n \} \cong \{ \ell_1 = e'_1, \dots, \ell_n =$
 916 $e'_n \})$.

917 **LEMMA 6.3.** *If $e_1 \cong e_2 : \tau$, $e_1 \Rightarrow v_1$, and $e_2 \Rightarrow v_2$, then for all patterns p where $p :: \tau \dashv \Gamma$, it is the
 918 case that either $v_1 \parallel p \dashv B$ and $v_2 \parallel p \dashv B$ or $v_1 \not\parallel p$ and $v_2 \not\parallel p$. Proof: by induction on $e_1 \cong e_2 : \tau$. If
 919 $\tau = \tau_1 \rightarrow \tau_2$ then by inversion of $p :: \tau \dashv \Gamma$, p must either be a wildcard or a variable. Then by **MATCH₁**
 920 and **MATCH₂**, we have that $v_1 \parallel p \dashv B$ and $v_2 \parallel p \dashv B$. If τ is not an arrow type, then we conclude by
 921 **EQ₂**.*

By conjunction and that all the Γ'_i are disjoint, we have that for all $i \in [n]$, $\forall_{\Gamma'_i}^{\text{val}} \sigma_i$. 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 $[\vec{v}/\vec{x}]e_i \Rightarrow v_i$ and $[\vec{v}/\vec{x}]e'_i \Rightarrow v'_i$ 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 $v_i \parallel p_i \dashv B_i$ and $v'_i \parallel p_i \dashv B_i$ or $v_i \not\parallel p_i$ and $v'_i \not\parallel p_i$.

To appeal to EQ₂, 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 $i \in [n]$ $v_i \parallel p_i \dashv B_i$ and $v'_i \parallel p_i \dashv B_i$, by MATCH₃ we have $\{\ell_1 = v_1, \dots, \ell_n = v_n\} \parallel p \dashv B_1 \dots B_n$ and $\{\ell_1 = v'_1, \dots, \ell_n = v'_n\} \parallel p \dashv B_1 \dots B_n$.
- In the case that there is some $i \in [n]$ where $v_i \not\parallel p_i$ and $v'_i \not\parallel p_i$, by MATCH₄ we have $\{\ell_1 = v_1, \dots, \ell_n = v_n\} \not\parallel p$ and $\{\ell_1 = v'_1, \dots, \ell_n = v'_n\} \not\parallel p$.

Since in all cases either $\{\ell_1 = v_1, \dots, \ell_n = v_n\} \parallel p \dashv B$ and $\{\ell_1 = v'_1, \dots, \ell_n = v'_n\} \parallel p \dashv B$ or $\{\ell_1 = v_1, \dots, \ell_n = v_n\} \not\parallel p$ and $\{\ell_1 = v'_1, \dots, \ell_n = v'_n\} \not\parallel p$, by EQ₂, we may conclude

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

ISO_{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 $[\vec{v}/\vec{z}] \left(\forall_{\Gamma'}^{\text{val}} \sigma \right)$. It must be shown that $[\vec{v}/\vec{z}](x e_1 \cong e_2)$.

By the inductive hypothesis we have

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

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

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

By assumption, we already have $[w/y][\vec{v}/\vec{z}] \left(\forall_{\Gamma'}^{\text{val}} \sigma \right)$. 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 $x e_1 \Rightarrow w$, 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 "+", "-", "*", "≤", "<", "≥", 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	$\text{int list list} \rightarrow \text{int list}$	concat takes a list of int lists and returns their concatenation without using the built-in “@” function
prefixSum	$\text{int list} \rightarrow \text{int list}$	prefixSum replaces each i -th element in an int list with the sum of the list’s first $i + 1$ elements
countNonZero	$\text{int tree} \rightarrow \text{int}$	countNonZero takes an int tree T and returns the number of nonzero nodes in T
quicksort	$(\text{'a * 'a} \rightarrow \text{order}) * \text{'a list} \rightarrow \text{'a list}$	quicksort implements the quicksort algorithm
slowDooop	$(\text{'a * 'a} \rightarrow \text{order}) * \text{'a list} \rightarrow \text{'a list}$	slowDooop takes a comparison function and uses it to remove all duplicates in a list
differentiate	$(\text{int} \rightarrow \text{real}) \rightarrow (\text{int} \rightarrow \text{real})$	differentiate differentiates a polynomial that is represented with the type $\text{int} \rightarrow \text{real}$
integrate	$(\text{int} \rightarrow \text{real}) \rightarrow \text{real} \rightarrow (\text{int} \rightarrow \text{real})$	integrate takes a polynomial p and a real c and returns the antiderivative of p with constant of integration c
treefoldr	$(\text{'a * 'b} \rightarrow \text{'b}) \rightarrow \text{'b} \rightarrow \text{'a tree} \rightarrow \text{'b}$	(treefoldr g init T) returns (foldr g init L) where L is the inorder traversal of T
treeReduce	$(\text{'a * 'a} \rightarrow \text{'a}) \rightarrow \text{'a} \rightarrow \text{'a tree} \rightarrow \text{'a}$	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	$(\text{'a} \rightarrow \text{bool}) \rightarrow (\text{'a * 'a} \rightarrow \text{bool}) \rightarrow \text{'a shrub} \rightarrow \text{int} \rightarrow (\text{'a list} \rightarrow \text{'b}) \rightarrow (\text{unit} \rightarrow \text{'b}) \rightarrow \text{'b}$	(findN p eq T n s k) returns s [x_1, \dots, x_n] where [x_1, \dots, x_n] are the leftmost values for T such that for all i from 1 to n , p x_i returns true and the x_i ’s are eq-distinct. (findN p eq T n s k) returns $k()$ if no such [x_1, \dots, x_n] exist
sat	$\text{prop} \rightarrow ((\text{string} * \text{bool}) \text{list} \rightarrow \text{'a}) \rightarrow (\text{unit} \rightarrow \text{'a}) \rightarrow \text{'a}$	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	$\text{'a list} \rightarrow (\text{'a list} \rightarrow \text{bool}) \rightarrow (\text{'a list} \rightarrow \text{bool}) \rightarrow \text{bool}$	(findPartition 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
slowDooop	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
slowDooop	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

1177 which A to use when returning $s(A)$. So for this task, two correct submissions could produce
1178 different outputs.

- 1179 • *The two programs use different recursive helper functions:* we found cases where equivalence
1180 classes were distinguished by the structure of the helper functions students created. Since our
1181 current inference rules do not consider these cases, ZEUS fails to recognize that two programs
1182 are equivalent if they use recursive helper functions with different input structures.
- 1183 • *The two recursive programs use different base cases:* our algorithm's treatment of fixed points
1184 causes it to never peer into a recursive call. Our algorithm's treatment of case expressions
1185 causes it to only recognize two expressions as equivalent if they handle all inputs in basically
1186 the same way. Together, these have the implication that when one expression treats a certain
1187 input as a base case while the other expression treats it as a recursive case, then the algorithm
1188 will be unable to recognize the expressions as equivalent.
- 1189 • *One of the programs uses built-in Standard ML functions:* seven out of the twelve tasks involve
1190 list manipulation operations. For instance, the top five tasks with the largest number of
1191 equivalence classes (`sat`, `findPartition`, `findN`, `quicksort`, and `prefixSum`) correspond to tasks
1192 that involve list manipulation. Many of the submissions for these tasks use built-in Standard
1193 ML functions for list reversal or list concatenation. We did not use a theory of list structures
1194 in our SMT Solver, so we were only able to recognize two expressions as equivalent if they
1195 used these built-in functions on the same input inputs and order or if they did not use these
1196 built-in functions at all.

1197 We note that even with the current limitations, ZEUS already shows that it can efficiently cluster
1198 the majority of the submissions into a few equivalence classes. Also, ZEUS could be extended by
1199 adding additional inference rules or support for additional SMT theories that would allow the
1200 identification of equivalent programs that are currently missed by ZEUS.

1201 8 RELATED WORK

1202 Proving that two problems are equivalent is a well-studied topic and has many applications ranging
1203 from hardware equivalence [Berman and Trevillyan 1989], compiler optimizations [Zuck et al. 2002],
1204 to program equivalence [Godlin and Strichman 2009]. However, the use of program equivalence
1205 for grading programming assignments is scarce [Kaleeswaran et al. 2016]. In this section, we cover
1206 related work from program equivalence and automatic grading that is closer to our approach.
1207

1208 8.1 Program Equivalence

1209 *Program Verification.* The problem of program equivalence can be reduced to a verification
1210 problem by showing that both programs satisfy the same specification. For instance, model-checking
1211 techniques [Clarke et al. 2004, 2001] can be used to show that two C programs satisfy the same
1212 specification. This specification can be written to ensure that for the same input, the programs are
1213 equivalent if they always produce the same output. Fedukovich et al. [Fedukovich et al. 2016]
1214 present techniques for proving that two similar programs have the same property rather than being
1215 equivalent. Their approach requires formally verifying one of the programs and using this proof to
1216 check the validity of the property in the other program by establishing a coupling between the two
1217 programs. A similar approach can also be done for functional programs. For instance, one could
1218 write a formal specification of the functionality of a program in Why3ML [Bobot et al. 2015]. We
1219 tried this approach by writing a formal specification for programming assignments for the function
1220 `concat`, however, the Why3 framework [Bobot et al. 2015] was not able to prove that the program
1221 satisfied the specification. In general, proving the program equivalence concerning a specification
1222 is a more challenging task than the one we address in this paper since we can take advantage of
1223 program structure to prove that they are equivalent.
1224
1225

1226 *Regression Verification.* In regression verification [Felsing et al. 2014; Godlin and Strichman 2009],
1227 the goal is to prove that two versions of a program are equivalent. One approach is to transform
1228 loops in programs to recursive procedures and to match the recursive calls in both programs and
1229 abstract them via uninterpreted functions [Godlin and Strichman 2009]. Other approaches, use
1230 invariant inference techniques to prove the equivalence of programs with loops [Felsing et al. 2014].
1231 By using these techniques, one can encode the two versions of the program into Horn clauses and
1232 use constraint solvers to automatically find certain kinds of invariants. Alternatively, one can also
1233 use symbolic execution and static analysis to generate summaries of program behaviors that capture
1234 the modifications between the programs. These summaries can be encoded into logical formulas
1235 and their equivalence can be checked using SMT solvers [Backes et al. 2013]. Our approaches
1236 also consider that student submissions are similar but they are not different versions of the same
1237 program. Even though we do not use any invariant generation techniques, this is orthogonal to our
1238 approach and could increase the number of equivalent classes detected for recursive programs.

1239
1240 *Contextual Equivalence.* There is a broad set of work that targets contextual equivalence for
1241 functional programs. Approaches based on step-indexed logical relations [Ahmed et al. 2009; Ahmed
1242 2006; Dreyer et al. 2009] or on bisimulations [Hur et al. 2012; Koutavas and Wand 2006; Sumii and
1243 Pierce 2005] have been used to prove context equivalence of functional programs with different
1244 fragments of ML that often include finite datatypes and integer references. While these approaches
1245 are more theoretical and focus on functional programs with state, we do not support state but can
1246 handle pattern matching which is crucial for a practical tool to cluster programming assignments
1247 of introductory functional courses. The closest approach to ours is the one recently presented by
1248 Jaber [Jaber 2020]. Jaber presents techniques for checking the equivalence of OCaml programs with
1249 state. His approach focuses in particular on contextual equivalence and developing a framework in
1250 which references can be properly accounted for. Our approach neglects references, as we require
1251 programs to be purely functional, but includes a more comprehensive treatment of datatypes. We
1252 attempted to compare our ZEUS's performance against Jaber's SYTECI prototype, but unfortunately,
1253 all of our benchmarks included datatypes that were not supported by the available prototype.

1254 8.2 Automatic grading

1255
1256 *Clustering similar assignments.* To help instructors to grade programming assignments, several
1257 automatic techniques have been proposed to cluster similar assignments into buckets with the
1258 purpose of giving automatic feedback [Gulwani et al. 2018; Kaleeswaran et al. 2016; Pu et al. 2016;
1259 Wang et al. 2018]. Our approach differs from these since our goal is not to replace the instructor or
1260 to fully automate the grading but rather to use ZEUS as a grading assistant with formal guarantees.

1261 CLARA [Gulwani et al. 2018] cluster correct programs and selects a canonical program from
1262 each cluster to be considered as the reference solution. In this approach, a pair of programs p_1 and
1263 p_2 are said to be dynamic equivalent if they have the same control-flow and if related variables
1264 in p_1 and p_2 always have the same values, in the same order, during the program execution on
1265 the same inputs. In contrast, our approach has a stronger notion of equivalence since we do not
1266 depend on dynamic program analysis. CodeAssist [Kaleeswaran et al. 2016] clusters submissions
1267 for dynamic programming assignments by their solution strategy. They consider a small set of
1268 features and if two programs share these features then they are put in the same cluster. Other
1269 clustering approaches are based on deep learning techniques [Pu et al. 2016] and also provide no
1270 formal guarantees about the quality of the clustering. SemCluster [Perry et al. 2019] improves
1271 upon other clustering techniques by considering semantic program features. They use control flow
1272 features and data flow features to represent each program and merge this information to create a
1273 program feature vector. K-means clustering is used to cluster all programs based on the program
1274

1275 feature vectors. Even though there are no formal guarantees for the equivalence of programs in
1276 each cluster, experimental results [Perry et al. 2019] show that the number of clusters found by
1277 SemCluster is much smaller than competitive approaches.

1278 *Automatic repair.* AutoGrader [Singh et al. 2013] takes as input a reference solution and an error
1279 model that consists of potential corrections and uses constraint solving techniques to find a mini-
1280 mum number of corrections that can be used to repair the incorrect student solution. Sarfgen [Wang
1281 et al. 2018] uses a three-stage algorithm based on search, align, and repair. It starts by searching for
1282 a small number of correct programs that can be used to repair the incorrect submission and have
1283 the same control-flow structure. Next, they compute a syntactic distance between those programs
1284 using an embedding of ASTs into numerical vectors. These programs are then aligned and the
1285 differences between aligned statements can suggest corrections that can be repaired automatically.

1286 Automatic repair is better suited for Massive Open Online Courses where a fully automated
1287 method is needed, while our approach is better suited for large, in-person courses, where the
1288 feedback of instructors can be more beneficial. The feedback returned by automatic repair tools is
1289 limited to changes in the code, while our approach is meant to assist instructors to provide more
1290 detailed feedback for students. Each equivalent class will have specific comments that are more
1291 helpful to the student than a repaired version of their submission. Moreover, while our approach can
1292 be used for both correct and incorrect submissions, automatic repair is only useful to fix incorrect
1293 submissions and cannot give any feedback for different implementations of correct submissions.

1294 *Formal guarantees.* Liu et al. [Liu et al. 2019] proposes to automatically determine the correctness
1295 of an assignment against a reference solution. Instead of using test cases, they use symbolic
1296 execution to search for semantically different execution paths between a student’s submission and
1297 the reference solution. If such paths exist, then the submission is considered incorrect and feedback
1298 can be provided by using counterexamples based on path deviations. Our approach is not based on
1299 symbolic execution but instead uses inference rules to derive a formula for which both student
1300 submissions are equivalent if and only if they have the same structure and the observable behavior.

1301 CodeAssist [Kaleeswaran et al. 2016] checks equivalence of a candidate submission from a cluster
1302 with a correct solution of that cluster that has been previously validated by an instructor. They
1303 exploit the fact of just handling dynamic programming assignments to establish a correspondence
1304 between variables and control locations of the two programs. Using this correspondence, they can
1305 encode the problem into SMT and prove program equivalence. Our approach is more general since
1306 our inference rules simulate relationships between expressions of the two programs and can be
1307 applied to several problem domains and not just dynamic programming assignments.

1308 9 CONCLUSION

1309 We present techniques for checking for equivalence between purely functional programs. Guided
1310 by inference rules that inform needed equivalences between two programs’ subexpressions, our
1311 approach simultaneously deconstructs the expressions being compared to build up a formula that
1312 is valid only if the expressions are equivalent. We prove the soundness of our approach: if our
1313 approach takes in two expressions of the same type and outputs a valid formula, then the two
1314 expressions are equivalent. We implement our approach and show that it can assist grading by
1315 clustering over 4,000 real student code submissions from an introductory functional programming
1316 class taught at the undergraduate level.

1317 ACKNOWLEDGMENTS

1318 This work was partially funded by National Science Foundation (Grants CCF-1901381, CCF-1762363,
1319 and CCF-1629444).

REFERENCES

- 1324
1325 Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proc. Symposium*
1326 *on Principles of Programming Languages*. ACM, 340–353.
- 1327 Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proc. European*
1328 *Symposium on Programming*. Springer, 69–83.
- 1329 John D. Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression Verification Using Impact Summaries.
1330 In *Proc. International Symposium Model Checking Software*. Springer, 99–116.
- 1331 C Leonard Berman and Louise H Trevillyan. 1989. Functional comparison of logic designs for VLSI circuits. In *Proc.*
1332 *International Conference on Computer-Aided Design*. IEEE, 456–459.
- 1333 François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2015. Let’s verify this with Why3. *Int. J.*
1334 *Softw. Tools Technol. Transf.* 17, 6 (2015), 709–727.
- 1335 Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *Proc. International*
1336 *Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.
- 1337 Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability
1338 Solving. *Formal Methods Syst. Des.* 19, 1 (2001), 7–34.
- 1339 Joshua Clune, Vijay Ramamurthy, Ruben Martins, and Umut A. Acar. 2020. Program Equivalence for Assisted Grading of
1340 Functional Programs (Extended Version). *CoRR* (2020).
- 1341 Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. International Conference on*
1342 *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
- 1343 Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *Proc. Annual Symposium on*
1344 *Logic in Computer Science*. IEEE Computer Society, 71–80.
- 1345 Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2016. Property Directed Equivalence via Abstract Simulation.
1346 In *Proc. International Conference Computer-Aided Verification*. Springer, 433–453.
- 1347 Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression
1348 verification. In *Proc. International Conference on Automated Software Engineering*. ACM, 349–360.
- 1349 Benny Godlin and Ofer Strichman. 2009. Regression verification. In *Proc. Design Automation Conference*. ACM, 466–471.
- 1350 Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated clustering and program repair for introductory
1351 programming assignments. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*.
1352 ACM, 465–480.
- 1353 Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical
1354 relations. In *Proc. Symposium on Principles of Programming Languages*. ACM, 59–72.
- 1355 Guilhem Jaber. 2020. SyTeCi: automating contextual equivalence for higher-order programs with references. *PACMPL* 4,
1356 *POPL* (2020), 59:1–59:28.
- 1357 Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback
1358 generation. In *Proc. International Symposium on Foundations of Software Engineering*. ACM, 739–750.
- 1359 Vasileios Koutavas and Mitchell Wand. 2006. Small bisimulations for reasoning about higher-order imperative programs. In
1360 *Proc. Symposium on Principles of Programming Languages*. ACM, 141–152.
- 1361 Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. 2019. Automatic grading of programming assignments: an approach
1362 based on formal semantics. In *Proc. International Conference on Software Engineering: Software Engineering Education and*
1363 *Training*. IEEE / ACM, 126–137.
- 1364 David Mitchel Perry, Dohyeong Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: clustering of imperative
1365 programming assignments based on quantitative semantic features. In *Proc. ACM SIGPLAN Conference on Programming*
1366 *Language Design and Implementation*. 860–873.
- 1367 Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. sk_p: a neural program corrector for
1368 MOOCs. In *Proc. International Conference on Systems, Programming, Languages and Applications: Software for Humanity*.
1369 ACM, 39–40.
- 1370 Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory
1371 programming assignments. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*.
1372 ACM, 15–26.
- 1373 Eijiro Sumii and Benjamin C. Pierce. 2005. A bisimulation for type abstraction and recursion. In *Proc. Symposium on*
1374 *Principles of Programming Languages*. ACM, 63–74.
- 1375 Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory
1376 programming exercises. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM,
1377 481–495.
- 1378 Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. 2002. VOC: A translation validator for optimizing compilers.
1379 *Electronic notes in theoretical computer science* 65, 2 (2002), 2–18.