Exploring Program Equivalence as a Means of Comparing Definitional Interpreters

Ruben Backx Casper Bach Poulsen Cas van der Rest

Delft University of Technology
Research Project - Q4 2020 / 2021

Abstract

Grading and giving feedback to student submissions automatically is becoming more and more necessary with an increasing amount of students. To verify the correctness of student-written definitional interpreters, a program equivalence approach has been implemented, improved, and extended with new rules to make it more suited specifically for verifying interpreters. This approach is able to soundly recognise two different interpreters as equivalent. Interpreters can thus be compared to a correct interpreter to verify their correctness or be grouped with equivalent interpreters to be graded in batches. Using program equivalence in combination with other verification approaches can improve the process of giving feedback to students and help build up a collection of common errors made by students.

1 Introduction

In recent times, as the amount of students has increased, so has the demand for more efficient automated grading. The course Concepts of Programming Languages at the Delft University of Technology is no exception. In this course, students write definitional interpreters, i.e. interpreters for a language defined in the same metalanguage as the interpreters. These interpreters are compared to the reference interpreters using an ever-growing amount of tests. Every year, students find ways of passing all tests with a wrong interpreter. The need to manually come up with tests for new edge cases would be reduced if student interpreters could be compared to the reference interpreters automatically.

There has been some research into comparing student submissions for programming assignments. Examples are Gulwani et al. [1], who cluster python programs and correct errors in them to assist in grading, and Jaber [2], who automatically determines whether higher-order programs are equivalent as long as they do not have recursion. Another approach is that by Clune et al. [3]. They convert programs to a logical formula that will be satisfiable only if the programs are equivalent. Student submissions can subsequently be grouped into 'buckets' of equivalent submissions for batch grading or, in the case that they are in a correct 'bucket', to be labeled as correct. Since this approach requires programs to be written in a purely functional language, it might be more suited for comparing definitional interpreters, as those are most commonly written in functional languages. However, it has some limitations. To name a few: recursive functions with a different base case are not recognised as equivalent, programs that use built-in functions are not always recognised as equivalent, and it does not have support for strings nor for lists.

How effective is program equivalence for comparing definitional interpreters? In the rest of this paper we explore this thoroughly:

- In section 3 we explain how we implemented list and string support.
- We introduce pre- and post-processing techniques to make program equivalence more reliable in section 3.
- We create an additional set of formula generation rules, specifically suited for comparing interpreters and prove their soundness in section 3.

- The effectiveness of these new additions and program equivalence in general needs to be examined. We set up an experiment comparing the effectiveness on different classes of interpreters and discuss the results in section 4.
- No approach is perfect, so in section 5 and 6 we discuss possible flaws and future work.

2 Methodology

A definitional interpreter can be many times more complicated than the programs compared in Clune et al. First of all, case matches might contain a dozen or more cases, with possibly deeper nested case matches inside of them. Secondly, the main recursive `interpret` function is called extensively in multiple equally valid places. An interpreter might additionally make use of a number of helper function. Finally, strings and lists are fundamental for interpreters with variable or environment support. This all might pose a problem when using the program equivalence method to compare definitional interpreters.

On the other hand, program equivalence is currently able to recognise equivalent programs when they are structurally similar. Because interpreters written in a functional style often are structurally almost identical, program equivalence seems like a suitable approach to verify them. The restriction, that programs should have recursive calls in similar places with the same arguments, lest they be more likely to not be recognised as equivalent, should additionally not be a problem for interpreters. Take for example the expression $l + r$. Both *interp l* and *interp r* will have to be called, otherwise the interpreter *should* be recognised as non-equivalent to a correct one. The equivalence of smaller parts of functional programs is even decidable, as long as they can be converted to simply typed lambda calculus [4].

The testing of the program equivalence method was split up into several steps. Before anything could be done, an implementation of the method was needed. We created a new implementation containing all the rules from the Clune et al. paper. We chose to do this rather than modify the implementation Clune et al. provide, mainly because modification is easier if the implementation is written with modification in mind. Then, support for strings and lists was added. The details of this are described in section 3. 120 different interpreters, both with and without errors, were written next. The interpreters were labeled according to the 'bucket' they should belong to, i.e. equivalent interpreters were given the same label. The exact distribution will be described in the section 4.

In the following step the interpreters were compared using the unmodified program equivalence approach. This was done to provide a baseline to which a modified program equivalence approach, more suited for definitional interpreters, could be compared. Additionally, these results have provided insight in what improvements to make.

After improvements had been made, the interpreters were once again compared using the same method. Since the insight the original results provided was used to make improvements, it could be the case that only the mistakes in the original interpreters will be recognised with the modified program equivalence implementation. Therefore another 120 interpreters, with new types and combinations of errors, were written using the same insight. The full approach used to overcome this selection bias can be found in section 5. The last set of interpreters was compared just like the other two to obtain the final set of results.

3 Equivalence Implementation and Improvement

The implementation of the program equivalence approach can be divided into three phases: conversion, processing, and formula generation. Haskell code was converted into a smaller language based on the one in the Clune et al. paper. The language remains largely unmodified save the addition of nil, cons, concatenation, and the *char* type. The syntax of this language can be found in figure 1. Since a Haskell program need not be fully explicitly typed, parts where a type is missing get assigned a fresh type variable instead. We added a type reconstruction algorithm [5] to be run as the first step of the

processing phase. This works as long as there is enough information to reconstruct a concrete type. It would not work on `\x -> x` for example, as its type can only be reconstructed as $a \to a$. The processing phase also performs a type check on the two programs being compared to ensure they have the same type. Afterwards, a process of formula generation takes place based on a set of formula generation rules, which make use of term judgement, freshening, and equating bindings rules. We added new rules to support the additions to the language. Figure 2 and 3 define new typing rules for patterns and expressions respectively. New rules for pattern matching and the dynamic semantics are defined in figure 4 and 5. New rules for term judgement, freshening, equating bindings, and formula generation are defined in figure 6, 7, 8, and 9. An explanation about the notation used in the figures can be found in appendix A. Although an understanding of the original rules and definitions not mentioned in this paper is welcomed, it is not necessary to understand the contributions. After a formula is generated, it gets converted to bindings for an SMT solver, like Z3 [6], which then attempts to find a counter-example to the formula.

We treat strings as lists of characters from conversion up to formula generation. In the formula generation phase, terms of type $[char]$ are converted to strings instead of lists. Lists can be handled in one of two ways. A list can be converted to a built-in list or to a data type with constructors `Nil` and `Cons`. We chose the latter approach to ease implementation. This way no extra logic specifically for lists is needed.

We added six new rules to improve equivalence recognition. These can be found in figure 10. The $\text{ISO}_{\text{commutative}}$ rules are designed to recognise equivalence between two operations that have their operands flipped. They only apply to the operators $+$, $*$, $=$, and $\neq$. These rules simply compare the left hand side to the right hand side of the other operation in addition to comparing left hand side and right hand side operands. The other two rules attempt to peer deeper into a case analysis expression to find potential recursive calls. Originally, the expression matched on in a case analysis had to be a term for the case analysis to be checked against an arbitrary expression. Doing this for every expression would be too computationally intensive, but only applying this tactic to specific case matches does not impact performance too much.

To increase the chances two case analyses get recognised as equivalent, we added a small amount of processing to be done before formula generation to ensure the cases are in a predictable order. First, a check is performed such that only cases that can safely be reordered are considered. Then the cases are sorted. This can apply to matches on injections, lists, constants, or a combination of those with matches on records. For records, the cases are sorted using a radix sort.

We added additional logic directly after formula generation, since SMT solvers do not understand all terms. Specifically, the wildcard term cannot be passed directly to most SMT solvers. Therefore extraction of nested wildcards has been added. When a nested wildcard is encountered, it will be replaced by a fresh variable. That variable will be set equal to the value the wildcard originally occupied. For example, $Just \cdot \{\ldots, \ell_i = \_, \ldots\} \equiv t$ is converted to $Just \cdot \{\ldots, \ell_i = x, \ldots\} \equiv t \wedge x \equiv \text{getJust}(t) \cdot \ell_i$.

We have not only made improvements in the formula generation stage, but also in the conversion stage. To achieve the best results, programs should be converted to result in as similar programs as possible. Smaller, less deep programs seem to provide better results as well. To create smaller programs, functions without recursive calls are converted to simple lambdas. To create more similar programs, if statements are always converted to their most 'non-negated' form. This means the condition $e_1 = e_2$ will not change, but $e_1 \neq e_2$ will be changed to $e_1 = e_2$ with swapped branches.

The rules introduced should be sound to preserve the soundness of the original program equivalence approach, i.e. if two programs are recognised as equivalent, they are definitely existentially equivalent. The set of original rules has been proven sound in Clune et al., so we only prove the soundness of the new rules. Explanation about the notation used in the proof can also be found in appendix A.

base types	b	::=	$int \mid boolean \mid char$	
types	τ	::=	b	base type
		$\mid$	δ	data type
		$\mid$	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$	product type
		$\mid$	$\tau_1 \to \tau_2$	function type
		$\mid$	$[\tau]$	list type
injection labels	i	::=	$\text{label}_1 \mid \text{label}_2 \mid ...$	
patterns	p	::=	$\_$	wildcard pattern
		$\mid$	x	variable pattern
		$\mid$	$\{\ell_1 = p_1, \ldots, \ell_n = p_n\}$	record pattern
		$\mid$	x as p	alias pattern
		$\mid$	c	constant pattern
		$\mid$	$i \cdot p$	injection pattern (with argument)
		$\mid$	i	injection pattern (without argument)
		$\mid$	$[]_\tau$	nil pattern
		$\mid$	$p_1 : p_2$	cons pattern
primitive operations	o	::=	$+ \mid - \mid * \mid < \mid > \mid \leq \mid \geq$	
		$\mid$	$=_\tau \mid \neq_\tau \mid +\!+$ (concat)	
expressions	e	::=	c	constant
		$\mid$	x	variable
		$\mid$	$\{\ell_1 = e_1, \ldots, \ell_n = e_n\}$	record
		$\mid$	$e \cdot \ell_i$	projection
		$\mid$	$i \cdot e$	injection (with argument)
		$\mid$	i	injection (without argument)
		$\mid$	`case` e $\{p_1.e_1 \mid \ldots \mid p_n.e_1\}$	case analysis
		$\mid$	$\lambda x.e$	abstraction
		$\mid$	$e_1\ e_2$	application
		$\mid$	`fix` x `is` e	fixed point
		$\mid$	o	primitive operation
		$\mid$	$[]_\tau$	nil
		$\mid$	$e_1 : e_2$	cons

Figure 1: The syntax of the smaller language

$$\frac{}{[]_\tau :: [\tau] \dashv}\ \textsc{Pat}_{\text{list1}} \qquad \frac{p_1 :: \tau \dashv \Gamma_1 \quad p_2 :: [\tau] \dashv \Gamma_2}{p_1 : p_2 :: [\tau] \dashv \Gamma_1\Gamma_2}\ \textsc{Pat}_{\text{list2}}$$

Figure 2: The new pattern typing rules

$$\frac{}{\Gamma \vdash []_\tau : [\tau]}\ \textsc{Ty}_{\text{list1}} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash e_1 : e_2 : [\tau]}\ \textsc{Ty}_{\text{list2}}$$

Figure 3: The new expression typing rules

$$\frac{}{[]_\tau \mathbin{/\!/} []_\tau \dashv} \textsc{Match}_{\text{list1}} \qquad \frac{v \neq []_\tau}{v \mathbin{\backslash\!\backslash} []_\tau} \textsc{Match}_{\text{list2}} \qquad \frac{v_1 \mathbin{/\!/} p_1 \dashv B_1 \quad v_2 \mathbin{/\!/} p_2 \dashv B_2}{v_1{:}v_2 \mathbin{/\!/} p_1{:}p_2 \dashv B_1 B_2} \textsc{Match}_{\text{List3}}$$

$$\frac{v_1 \mathbin{\backslash\!\backslash} p_1}{v_1{:}v_2 \mathbin{\backslash\!\backslash} p_1{:}p_2} \textsc{Match}_{\text{list4}} \qquad \frac{v_2 \mathbin{\backslash\!\backslash} p_1}{v_1{:}v_2 \mathbin{\backslash\!\backslash} p_1{:}p_2} \textsc{Match}_{\text{list5}}$$

Figure 4: The new pattern matching rules

$$\frac{}{[]_\tau \text{ val}} \textsc{Dyn}_{\text{list1}} \qquad \frac{e_1 \mapsto e_1'}{e_1{:}e_2 \mapsto e_1'{:}e_2} \textsc{Dyn}_{\text{list2}}$$

$$\frac{e_1 \text{ val} \quad e_2 \mapsto e_2'}{e_1{:}e_2 \mapsto e_1{:}e_2'} \textsc{Dyn}_{\text{list3}} \qquad \frac{e_1 \text{ val} \quad e_2 \text{ val}}{e_1{:}e_2 \text{ val}} \textsc{Dyn}_{\text{list4}}$$

Figure 5: The new dynamic semantics

$$\frac{}{[]_\tau \text{ Term}} \textsc{Term}_{\text{list1}} \qquad \frac{t_1 \text{ Term} \quad t_2 \text{ Term}}{t_1{:}t_2 \text{ Term}} \textsc{Term}_{\text{list2}}$$

Figure 6: The new term judgement rules

$$\frac{}{\text{freshen } []_\tau.e \hookrightarrow []_\tau.e} \textsc{Freshen}_{\text{list1}} \qquad \frac{\text{freshen } p_1.e \hookrightarrow p_1'.e_1 \quad \text{freshen } p_2.e_1 \hookrightarrow p_2'.e_2}{\text{freshen } p_1{:}p_2.e \hookrightarrow p_1'{:}p_2'.e_2} \textsc{Freshen}_{\text{list2}}$$

Figure 7: The new freshening rules

$$\frac{}{\text{EB}([]_\tau.e_1, []_\tau.e_2) \hookrightarrow ([]_\tau.e_1, []_\tau.e_2)} \text{EB}_{\text{list1}}$$

$$\frac{\text{EB}(p_1^1.e_1, p_1^2.e_2) \hookrightarrow (p_1^3.e_1', p_1^4.e_2') \quad \text{EB}(p_2^1.e_1', p_2^2.e_2') \hookrightarrow (p_2^3.e_1'', p_2^4.e_2'')}{\text{EB}(p_1^1{:}p_2^1.e_1, p_1^2{:}p_2^2.e_2) \hookrightarrow (p_1^3{:}p_2^3.e_1'', p_1^4{:}p_2^4.e_2'')} \text{EB}_{\text{list2}}$$

Figure 8: The new equate bindings rules

$$\frac{\Gamma \vdash e_1 \overset{\sigma_1}{\Longleftrightarrow} e_1' : \tau \dashv \Gamma_1 \quad \Gamma \vdash e_2 \overset{\sigma_2}{\Longleftrightarrow} e_2' : [\tau] \dashv \Gamma_2}{\Gamma \vdash e_1{:}e_2 \overset{(\sigma_1 \wedge \sigma_2)}{\longleftrightarrow} e_1'{:}e_2' : [\tau] \dashv \Gamma_1 \Gamma_2} \textsc{Iso}_{\text{list1}}$$

Figure 9: The new formula generation rule

$$\frac{\begin{array}{c}\Gamma \vdash e_l \overset{\sigma_1}{\Longleftrightarrow} e_l' : \tau \dashv \Gamma_1 \quad \Gamma \vdash e_r \overset{\sigma_2}{\Longleftrightarrow} e_r' : \tau \dashv \Gamma_2 \\ \Gamma \vdash e_l \overset{\sigma_3}{\Longleftrightarrow} e_r' : \tau \dashv \Gamma_3 \quad \Gamma \vdash e_r \overset{\sigma_4}{\Longleftrightarrow} e_l' : \tau \dashv \Gamma_4\end{array}}{\Gamma \vdash e_l + e_r \overset{(\sigma_1 \wedge \sigma_2) \vee (\sigma_3 \wedge \sigma_4)}{\longleftrightarrow} e_l' + e_r' : \tau \dashv \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4} \text{ ISO}_{\text{commutative1}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_l \overset{\sigma_1}{\Longleftrightarrow} e_l' : \tau \dashv \Gamma_1 \quad \Gamma \vdash e_r \overset{\sigma_2}{\Longleftrightarrow} e_r' : \tau \dashv \Gamma_2 \\ \Gamma \vdash e_l \overset{\sigma_3}{\Longleftrightarrow} e_r' : \tau \dashv \Gamma_3 \quad \Gamma \vdash e_r \overset{\sigma_4}{\Longleftrightarrow} e_l' : \tau \dashv \Gamma_4\end{array}}{\Gamma \vdash e_l * e_r \overset{(\sigma_1 \wedge \sigma_2) \vee (\sigma_3 \wedge \sigma_4)}{\longleftrightarrow} e_l' * e_r' : \tau \dashv \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4} \text{ ISO}_{\text{commutative2}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_l \overset{\sigma_1}{\Longleftrightarrow} e_l' : \tau \dashv \Gamma_1 \quad \Gamma \vdash e_r \overset{\sigma_2}{\Longleftrightarrow} e_r' : \tau \dashv \Gamma_2 \\ \Gamma \vdash e_l \overset{\sigma_3}{\Longleftrightarrow} e_r' : \tau \dashv \Gamma_3 \quad \Gamma \vdash e_r \overset{\sigma_4}{\Longleftrightarrow} e_l' : \tau \dashv \Gamma_4\end{array}}{\Gamma \vdash e_l = e_r \overset{(\sigma_1 \wedge \sigma_2) \vee (\sigma_3 \wedge \sigma_4)}{\longleftrightarrow} e_l' = e_r' : \tau \dashv \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4} \text{ ISO}_{\text{commutative3}}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_l \overset{\sigma_1}{\Longleftrightarrow} e_l' : \tau \dashv \Gamma_1 \quad \Gamma \vdash e_r \overset{\sigma_2}{\Longleftrightarrow} e_r' : \tau \dashv \Gamma_2 \\ \Gamma \vdash e_l \overset{\sigma_3}{\Longleftrightarrow} e_r' : \tau \dashv \Gamma_3 \quad \Gamma \vdash e_r \overset{\sigma_4}{\Longleftrightarrow} e_l' : \tau \dashv \Gamma_4\end{array}}{\Gamma \vdash e_l \neq e_r \overset{(\sigma_1 \wedge \sigma_2) \vee (\sigma_3 \wedge \sigma_4)}{\longleftrightarrow} e_l' \neq e_r' : \tau \dashv \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4} \text{ ISO}_{\text{commutative4}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \text{ contains } e : \tau} \text{ CONTAINS}_{\text{refl}} \qquad \frac{\Gamma \vdash e_i \text{ contains } e : \tau}{\Gamma \vdash \{\ldots, \ell_i = e_i, \ldots\} \text{ contains } e : \tau} \text{ CONTAINS}_{\text{record}}$$

$$\frac{\Gamma \vdash e_1 \text{ contains } e : \tau}{\Gamma \vdash e_1\ e_2 \text{ contains } e : \tau} \text{ CONTAINS}_{\text{application}} \qquad \frac{\Gamma \vdash e' \text{ contains } e : \tau}{\Gamma \vdash i \cdot e' \text{ contains } e : \tau} \text{ CONTAINS}_{\text{injection}}$$

$$\frac{\begin{array}{c}y \text{ fresh} \quad \Gamma \vdash e_1 \text{ contains } f\ e_3 : \tau_2 \\ \Gamma, y : \tau_2 \vdash [y/(f\ e_3)]\texttt{case}\ e_1\ \{M\} \overset{\sigma}{\longleftrightarrow} [y/(f\ e_3)]e_2 : \tau \dashv \Gamma'\end{array}}{\Gamma \vdash \texttt{case}\ e_1\ \{M\} \overset{\sigma}{\longleftrightarrow} e_2 \dashv \Gamma'} \text{ ISO}_{\text{case6}}$$

$$\frac{\begin{array}{c}y \text{ fresh} \quad \Gamma \vdash e_2 \text{ contains } f\ e_3 : \tau_2 \\ \Gamma, y : \tau_2 \vdash [y/(f\ e_3)]e_1 \overset{\sigma}{\longleftrightarrow} [y/(f\ e_3)]\texttt{case}\ e_2\ \{M\} : \tau \dashv \Gamma'\end{array}}{\Gamma \vdash e_1 \overset{\sigma}{\longleftrightarrow} \texttt{case}\ e_2\ \{M\} \dashv \Gamma'} \text{ ISO}_{\text{case7}}$$

Figure 10: The new rules implemented after experimentation

Theorem 3.1. *The new* ISO *rules introduced in this paper are sound. That is, for any expressions e_1 and e_2*

$$\textit{if } \Gamma_{\textit{initial}} \vdash e_1 \overset{\sigma}{\Leftrightarrow} e_2 : \tau \dashv \Gamma' \textit{ and } \overset{\textit{val}}{\forall}_{\Gamma'} .\sigma$$
$$\textit{then } e_1 \cong e_2 : \tau$$

Proof. We prove the soundness of each rule by induction. We use the following induction hypothesis:

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

We will use the fact that the language enjoys referential transparency, meaning existential equivalence is closed under replacement of existentially equivalent sub-expressions.

$\text{ISO}_{\text{commutative1}}$: Let $\Gamma = \vec{x} : \vec{\tau}$ with arbitrary $\vec{v}$, where $\forall v_i \in \vec{v} \, (v_i : \tau_i \wedge v_i \text{ val})$.
Assume $[\vec{v}/\vec{x}] \left(\overset{\text{val}}{\forall}_{\Gamma_1,\Gamma_2,\Gamma_3,\Gamma_4} (\sigma_1 \wedge \sigma_2) \vee (\sigma_3 \wedge \sigma_4) \right)$.
We must show $[\vec{v}/\vec{x}](e_l + e_r \cong e'_l + e'_r)$.
By the induction hypothesis

$$\overset{\text{val}}{\forall}_{\Gamma} . \left(\text{if } \left(\overset{\text{val}}{\forall}_{\Gamma_1} .\sigma_1 \right) \text{ then } e_l \cong e'_l \right)$$

thus, since σ_1 does not contain any variables in Γ_2, Γ_3, or Γ_4

$$\text{if } [\vec{v}/\vec{x}] \left(\overset{\text{val}}{\forall}_{\Gamma_1} .\sigma_1 \right) \text{ then } [\vec{v}/\vec{x}] \left(e_l \cong e'_l \right)$$

The same goes for σ_2, σ_3, and σ_4, giving

$$\text{if } [\vec{v}/\vec{x}] \left(\overset{\text{val}}{\forall}_{\Gamma_2} .\sigma_2 \right) \text{ then } [\vec{v}/\vec{x}] \left(e_r \cong e'_r \right)$$

$$\text{if } [\vec{v}/\vec{x}] \left(\overset{\text{val}}{\forall}_{\Gamma_3} .\sigma_3 \right) \text{ then } [\vec{v}/\vec{x}] \left(e_l \cong e'_r \right)$$

$$\text{if } [\vec{v}/\vec{x}] \left(\overset{\text{val}}{\forall}_{\Gamma_4} .\sigma_4 \right) \text{ then } [\vec{v}/\vec{x}] \left(e_r \cong e'_l \right)$$

From this we can conclude

$$[\vec{v}/\vec{x}] \left(e_l \cong e'_l \right) \wedge [\vec{v}/\vec{x}] \left(e_r \cong e'_r \right)$$
$$\vee \, [\vec{v}/\vec{x}] \left(e_l \cong e'_r \right) \wedge [\vec{v}/\vec{x}] \left(e_r \cong e'_l \right)$$

By referential transparency

$$[\vec{v}/\vec{x}] \left(e_l + e_r \cong e'_l + e'_r \right)$$
$$\vee \, [\vec{v}/\vec{x}] \left(e_l + e_r \cong e'_r + e'_l \right)$$

Since $+$ is commutative $[\vec{v}/\vec{x}] \left(e'_l + e'_r \cong e'_r + e'_l \right)$, therefore

$$[\vec{v}/\vec{x}] \left(e_l + e_r \cong e'_l + e'_r \right)$$

$\text{ISO}_{\text{commutative2}}$: The proof is the same as in $\text{ISO}_{\text{commutative1}}$ with $+$ substituted for $*$.
$\text{ISO}_{\text{commutative3}}$: The proof is the same as in $\text{ISO}_{\text{commutative1}}$ with $+$ substituted for $=$.
$\text{ISO}_{\text{commutative4}}$: The proof is the same as in $\text{ISO}_{\text{commutative1}}$ with $+$ substituted for $\neq$.
$\text{ISO}_{\text{Case6}}$: Let $\Gamma = \vec{x} : \vec{\tau}$ with arbitrary $\vec{v}$, where $\forall v_i \in \vec{v} \, (v_i : \tau_i \wedge v_i \text{ val})$.
Assume $[\vec{v}/\vec{x}] \left(\overset{\text{val}}{\forall}_{\Gamma'} \sigma \right)$.
We must show $[\vec{v}/\vec{x}](\texttt{case}\ e_1\ \{M\} \cong e_2)$.
By the induction hypothesis

$$\overset{\text{val}}{\forall}_{\Gamma} . \left(\text{if } \left(\overset{\text{val}}{\forall}_{\Gamma'} .\sigma \right) \text{ then } [y/(f\ e_3)]\texttt{case}\ e_1\ \{M\} \cong [y/(f\ e_3)]e_2 \right)$$

We can assume $f\ e_3 \mapsto w$ for some w, such that $\Gamma \vdash w : \tau$ and w val, since we only need to prove soundness for valuable expressions. Since y is fresh

$$\text{if } [y/w][\vec{v}/\vec{x}] \left(\overset{\text{val}}{\forall}_{\Gamma'} .\sigma \right) \text{ then } [y/w][\vec{v}/\vec{x}] \left([y/(f\ e_3)]\texttt{case}\ e_1\ \{M\} \cong [y/(f\ e_3)]e_2 \right)$$

By assumption we conclude

$$[y/w][\vec{v}/\vec{x}] \left([y/(f\ e_3)]\texttt{case}\ e_1\ \{M\} \cong [y/(f\ e_3)]e_2 \right)$$

which is equivalent to

$$[\vec{v}/\vec{x}] \left([w/(f\ e_3)]\texttt{case}\ e_1\ \{M\} \cong [w/(f\ e_3)]e_2 \right)$$

Because $f\ e_3 \mapsto w$ and by referential transparency

$$[\vec{v}/\vec{x}] \left([(f\ e_3)/(f\ e_3)]\texttt{case}\ e_1\ \{M\} \cong [(f\ e_3)/(f\ e_3)]e_2 \right)$$

which is

$$[\vec{v}/\vec{x}]\,(\texttt{case}\ e_1\ \{M\} \cong e_2)$$

$\text{ISO}_{\text{Case7}}$: By symmetry and $\text{ISO}_{\text{Case6}}$.

□

4 Experimental

The initial set of test interpreters written consists of 120 interpreters in total, equally divided over over three categories. One category contains interpreters for pure lambda calculus, one contains type checkers, and the last category contains interpreters for a more complicated language. These languages are described in appendix B. Type checkers have been included as their structure is similar to interpreters and they are often written by students alongside interpreters. Every category consists of 20 correct and 20 incorrect interpreters. The 20 correct interpreters are only structurally different from each other. 10 of the incorrect interpreters have one or two mistakes. The other 10 are structurally different from one of the first 10.

To test the effectiveness, all interpreters are compared to all other interpreters from from same category. This results in 580 checks that should be negative and 200 checks that should be positive. The results of this for the program equivalence approach without any new rules added can be found in figure 12. True negatives and false positives have been included as a sanity check. If there would be false positives, one of the rules would be unsound. These results show the program equivalence approach is able to recognise some interpreters as equivalent, but some rules need to be added to make it more viable. By analysing the false negatives, we made a list of changes that without any new rules result in programs not being recognised as equivalent. Relevant rules can be found in figure 11.

Case ordering When two case analyses are being compared, it is helpful if the cases are in the same order. Case analyses with differently ordered cases cannot make use of the $\text{ISO}_{\text{case4}}$ and $\text{ISO}_{\text{case5}}$ rules effectively, as the freshen together judgements require the cases to be in the same order.

Nested patterns Since patterns are terms, they can appear nested in records and injections. Often a term is generated of the form $x \equiv \{\ell_1 = \_, \ldots\}$. The naive approach of converting the record directly to some value in the SMT solver does not work. The wildcard cannot be converted to a value. This term can also not be filtered out in simplification like $x \equiv \_$, which simplifies to *true*. This caused the equivalence checker to fail and assume the programs being compared are not equivalent.

Complex case analysis Comparing a case analysis that analyses an non-term expression, e.g. `case` $interp\ e\ \{\ldots\}$, means that $\text{ISO}_{\text{case1}}$ and $\text{ISO}_{\text{case2}}$ cannot be used. If the other expression is not a case analysis, no formula can be generated, causing the equivalence checker to assume the programs are not equivalent. These kinds of case analyses occur often in interpreters, increasing the chance that no formula is generated.

Nested case analyses Nested case analyses increase the likelihood of generating an equivalence discussed in the previous point, resulting in no formula being generated.

Commutative operations The application rules usually do not detect equivalence in commutative operations, e.g. $e_1 * e_2 \leftrightarrow e_3 * e_4$. A part of the term this generates is $\sigma_1 \wedge \sigma_2 \wedge * \equiv *$, where σ_1 and σ_2 are generated by $e_1 \Leftrightarrow e_3$ and $e_2 \Leftrightarrow e_4$ respectively. Ideally $e_1 \Leftrightarrow e_4$ and $e_2 \Leftrightarrow e_3$ would also be checked.

Negated conditions Two implementations that check the same condition cannot be recognised as equivalent when one of them is negated, e.g.

$$\begin{array}{l} \texttt{case}\ e_1 = e_2\ \{true.e_3 | false.e_4\} \\ \qquad\qquad \updownarrow \\ \texttt{case}\ e_1 \neq e_2\ \{true.e_4 | false.e_3\} \end{array}$$

Clearly these are equivalent, but no rule can recognise this.

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

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

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

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

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

Figure 11: Relevant original rules

The improvements described in section 3 have solved most of these problems in some way. Nested patterns and negated conditions have been completely solved by pattern extraction and better conversion. Case ordering is solved by case reordering except in the case of some record patterns. It might be possible to fix this as well with a more sophisticated reordering approach. Complex case analysis and commutative operations have been solved as a consequence of the new ISO_{case} rules and the $\text{ISO}_{\text{commutative}}$ rules. Nested case analysis is still a common problem, although it is no longer as frequent.

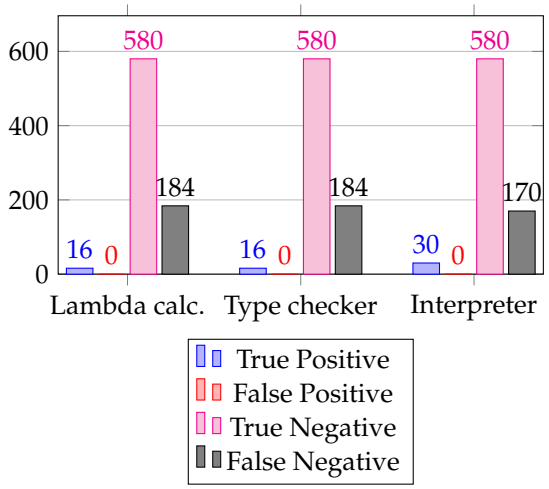

Figure 12: Results with no additional rules

The new results on the original set of interpreters can be found in figure 13. This shows an improvement over unmodified program equivalence. This improvement can also be seen on a new set of interpreters. These result can be found in figure 14. Depending on the type of interpreter about 25% to 65% of the tested modifications can be detected as equivalent.

To conclude the experimental results, we performed a small performance analysis. The averages of 10 different execution times of the unmodified and modified program equivalence approaches were taken. For all runs, two equivalent interpreters were compared. The modifications should not make the approach too slow in order for it to be practical, nor should the unmodified approach be slow. The results are as follows: 0.42 seconds for unmodified program equivalence and 0.69 seconds for modified program equivalence. These numbers do not differ much for smaller or larger interpreters, as the execution time largely depends on the amount of applied formula generation rules. This number is similar for a pair of similarly different interpreters. These results shows the improvements slow comparison down slightly, but two equivalent interpreters can still be compared in less than a second with and without improvements.

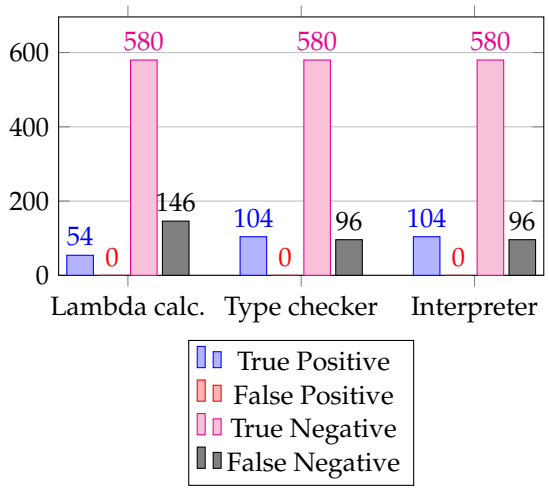

Figure 13: Results from the first set of interpreters with the full ruleset

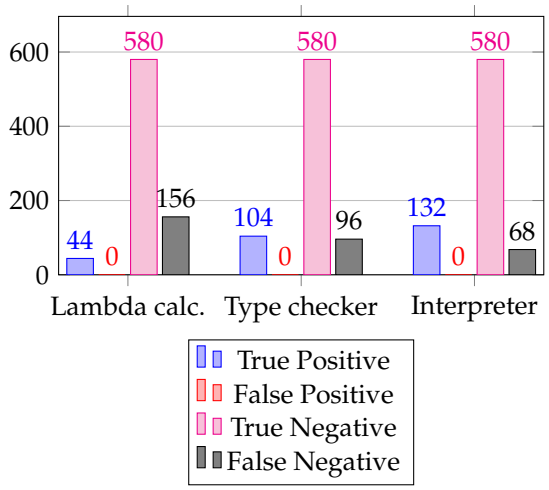

Figure 14: Results from the second set of interpreters with the full ruleset

5 Responsible Research

Any approach that automatically grades students should not be blindly trusted. The soundness of the program equivalence approach makes it such that the main concern is false negatives. This is why only interpreters for which a counter-example gives different results can be labeled as 'definitely different'. If no such counter-example is generated, the interpreters might simply be too structurally different. This can be of interest in grading, but should never be used to label a solution as incorrect.

One should also keep in mind the possibility that the program equivalence approach does not work well on interpreters written by students. It has been tested on interpreters with common mistakes students make when writing interpreters, but this is not a guarantee it will work in a real world situation. Hence why it is important all interpreters used to test be available. They have been made available open source at https://gitlab.ewi.tudelft.nl/cse3000-auto-test/test-suites.

Next, the possibility of a bug in the implementation needs to be discussed. It is likely a bug was introduced in the implementation used for this paper or will be introduced in any re-implementation in the future. To reduce the probability of this happening, any implementation should be extensively tested. The implementation used for this paper has been tested manually on many interpreters of different structures. Automated tests are also advisable for future implementations. Bugs can lead to more false negatives or even false positives, violating the soundness of the approach. During testing for this paper no such bug has been found. This does not mean there is none however. For full transparency the implementation used has also been made available open source at https://gitlab.ewi.tudelft.nl/cse3000-auto-test/programme-equivalence.

Lastly, to avoid selection bias, some precautions were taken. Although mentioned in this paper, the effectiveness on the original set of test interpreters should not be taken as the real effectiveness of the approach. The effectiveness on the second set of interpreters is more accurate. The first set of interpreters was used to gain insight on which rules should be added to make the program equivalence approach more suitable for interpreters. Creating two sets of interpreters would be useless however, if the same types of errors and structural changes were used in the new interpreters as well. This is why, if a rule was added that made the approach more effective for interpreters with a certain error or structural change, this change was not made in any of the new interpreters. Only when a change proved hard to detect, was it used in the new interpreters with the same frequency as in the original set. This likely still does not give an accurate depiction of the real world and therefore the results only serve as a crude approximation of the effectiveness of program equivalence to verify definitional interpreters.

6 Using Program Equivalence

Based on the experimental results, we can conclude that the program equivalence approach is at least suitable for automatically verifying student-written interpreters. Not only because the approach seems to recognise about half of the equivalent interpreters, but more importantly because it is sound. In a real world setup, student submissions could be compared to a reference interpreter. If an interpreter is then recognised as equivalent, it can be assumed that the student's interpreter is not only correct, but also likely structurally similar to the reference interpreter. Another possibility is to 'bucketise' student submissions. Interpreters can be compared to each other and equivalent interpreters can be grouped in the same 'bucket'. With this setup, manual or automatic feedback given to one student likely applies to all other students in the same bucket as well.

Something program equivalence lacks is the ability to generate a good counter-example for two non-equivalent interpreters. When two interpreters are non-equivalent, usually a counter-example is generated that gives a different result on the two interpreters. This is however not guaranteed. Additionally, it might be the case two interpreters are equivalent, just not recognised as such. Another problem is the fact that not all counter-examples are meaningful. Therefore it might be beneficial to pair the program equivalence approach with some

other verification approaches. One option is concolic or symbolic execution [7] [8]. The benefit of such an approach is that it can run on the smaller language the interpreters get converted to, easing implementation. Since these approaches generate their counter-examples by recording constraints against execution paths, the counter-examples are more likely to be useful. Alternatively, a property based testing tool, like QuickCheck, can be used [9] [10]. Property based testing tools make use of randomisation to generate counter-examples that violate some property. These counter-examples are usually large for more complex programs. This is why property based testing tools can employ 'shrinking' to make a counter-example as small as possible. Both these approaches are suitable to generate counter-examples for two non-equivalent interpreters. They might also be used to gain insight into why an interpreter is incorrect. Combining this with bucketising can be used to provide better feedback to groups of students at once or to collect common errors, which correspond to buckets with many incorrect submissions.

7 Conclusion

We have presented an approach to check the equivalence of definitional interpreters, based on earlier work about program equivalence. We have introduced new rules and steps in the process to make the approach more suited for definitional interpreters. The main question, whether program equivalence is suitable for verifying definitional interpreters, has been answered. We have proven the soundness of the approach and experimentally assessed its effectiveness. Without any other methods, we can already verify the correctness of an interpreter most of the time by comparing it to a reference interpreter. In combination with other methods, interpreters can with certainty be marked as incorrect as well.

The rules introduced provide better results, but the results are far from perfect and come at a performance cost. In the future, more rules could be discovered. However, their soundness and run time cost should be taken into consideration. Improving equivalence recognition might also be achieved by improving other steps in the process such as conversion and formula simplification. Another open question is whether program equivalence will prove effective in a real world setting; more experiments can be conducted on larger sets of interpreters.

References

[1] S. Gulwani, I. Radiček and F. Zuleger, 'Automated clustering and program repair for introductory programming assignments,' in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia PA USA: ACM, 11th Jun. 2018, pp. 465–480, ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192387. [Online]. Available: https://dl.acm.org/doi/10.1145/3192366.3192387 (visited on 22/04/2021).

[2] G. Jaber, 'SyTeCi: Automating contextual equivalence for higher-order programs with references,' *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–28, POPL Jan. 2020, ISSN: 2475-1421. DOI: 10.1145/3371127. [Online]. Available: https://dl.acm.org/doi/10.1145/3371127 (visited on 22/04/2021).

[3] J. Clune, V. Ramamurthy, R. Martins and U. A. Acar, 'Program equivalence for assisted grading of functional programs,' *Proc. ACM Program. Lang.*, 4(OOPSLA):171:1–171:29, Nov. 2020. DOI: 10.1145/3428239. [Online]. Available: https://doi.org/10.1145/3428239 (visited on 22/04/2021).

[4] B. C. Pierce, 'Logical relations and a case study in equivalence checking,' in *Advanced topics in types and programming languages*, Cambridge, Mass: MIT Press, 2005, pp. 223–244, ISBN: 978-0-262-16228-9.

[5] ——, 'Type reconstruction,' in *Types and programming languages*, Cambridge, Mass: MIT Press, 2002, pp. 317–336, ISBN: 978-0-262-16209-8.

[6] L. de Moura and N. Bjørner, 'Z3: An efficient smt solver,' in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[7] S.-H. You, R. B. Findler and C. Dimoulas, 'Sound and complete concolic testing for higher-order functions,' in *Programming Languages and Systems*, N. Yoshida, Ed., vol. 12648, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2021, pp. 635–663, ISBN: 978-3-030-72018-6 978-3-030-72019-3. DOI: 10.1007/978-3-030-72019-3_23. [Online]. Available: http://link.springer.com/10.1007/978-3-030-72019-3_23 (visited on 14/06/2021).

[8] A. D. Mensing, H. van Antwerpen, C. Bach Poulsen and E. Visser, 'From definitional interpreter to symbolic executor,' in *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection - META 2019*, Athens, Greece: ACM Press, 2019, pp. 11–20, ISBN: 978-1-4503-6985-5. DOI: 10.1145/3358502.3361269. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3358502.3361269 (visited on 14/06/2021).

[9] L. Lampropoulos, M. Hicks and B. C. Pierce, 'Coverage guided, property based testing,' *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–29, OOPSLA 10th Oct. 2019, ISSN: 2475-1421. DOI: 10.1145/3360607. [Online]. Available: https://dl.acm.org/doi/10.1145/3360607 (visited on 14/06/2021).

[10] K. Claessen and J. Hughes, 'QuickCheck: A lightweight tool for random testing of haskell programs,' in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming - ICFP '00*, Not Known: ACM Press, 2000, pp. 268–279, ISBN: 978-1-58113-202-1. DOI: 10.1145/351240.351266. [Online]. Available: http://portal.acm.org/citation.cfm?doid=351240.351266 (visited on 22/04/2021).

A Notation

Pattern typing In pattern typing $p :: \tau \dashv \Gamma$ means that pattern p can match the type τ and will bind the variables in Γ in the following expression. Γ can be omitted if the pattern binds no new variables.

Expression typing $\Gamma \vdash e : \tau$ can be read as: given an environment Γ, the expression e has type τ.

Pattern matching Similar to pattern typing $e /\!\!/ p \dashv B$ binds the variables in B in the following expression. Rather than matching a type, in this case, an expression e matches pattern p. $e \not/\!\!/ p$ indicates the expression does not match the pattern.

Dynamic semantics e val means the expression e is a value and cannot be interpreted further. $e \mapsto e'$ indicates that expression e can be interpreted in one step to e'. Similarly, $e \Mapsto v$ indicates e can be interpreted in many steps to v.

Term judgement To indicate an expression is a valid logical term e Term is used.

Freshening Freshening of a pattern p with its corresponding expression e is indicated as freshen $p.e \hookrightarrow p'.e'$. p' is the resulting pattern and e' is the resulting expression.

Equate bindings Two pattern-expression pairs $p_1.e_1$ and $p_2.e_2$ can be equated resulting in $p'_1.e'_1$ and $p'_2.e'_2$. This is indicated as $\mathrm{EB}(p_1.e_1, p_2.e_2) \hookrightarrow \mathrm{EB}(p'_1.e'_1, p'_2.e'_2)$

Formula generation Formula generation statements are of the form $\Gamma \vdash e_1 \overset{\sigma}{\leftrightarrow} e_2 : \tau \dashv \Gamma'$. Here e_1 and e_2 are the two expressions being checked for equivalence. Γ contains the variables such that $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$. The result is a new set of variables Γ', disjoint from Γ. The resulting logical formula is σ. If e_1 and e_2 are first reduced to weak head normal form, $\Gamma \vdash e_1 \overset{\sigma}{\Leftrightarrow} e_2 : \tau \dashv \Gamma'$ is written. In this paper, $e_1 \leftrightarrow e_2$ and $e_1 \Leftrightarrow e_2$ are used to indicate that e_1 and e_2 are being checked for equivalence.

Existential equivalence $e_1 \cong e_2 : \tau$ is used to indicate e_1 and e_2 have type τ and are existentially equivalent, i.e. they behave identically in any program. $: \tau$ may be omitted if the type is obvious or not important.

Substitution When an expression y is substituted by an expression x in e, $[x/y]e$ is used. A list of expressions $\vec{y}$ can be substituted by a list of expressions $\vec{x}$ as well. In that case, notation remains $[\vec{x}/\vec{y}]e$.

Validity The validity of σ is denoted $\overset{\text{val}}{\forall}_\Gamma .\sigma$. This means that for all $\vec{v}$ where $\Gamma \vdash v_i : \tau_i$ and v_i val for all $v_i \in \vec{v}$, $[\vec{v}/\vec{x}]\sigma$ holds.

Binary operators Although the language from figure 1 does not have infix operators, $e_1 \circ e_2$ is used to denote $(\circ\, e_1)\, e_2$, where $\circ$ is any binary operator. Additionally the τ from $=_\tau$ and $\neq_\tau$ may be omitted when the type of the operands is obvious or not important.

B Languages Used in Experimentation

Lambda Calculus

expressions	e	::=	x	variable
		|	$\lambda x.e$	abstraction
		|	$e_1\ e_2$	application

Interpreter

expressions	e	::=	n	integer
		|	x	variable
		|	$\lambda x.e$	abstraction
		|	$e_1\ e_2$	application
		|	$e_1 + e_2$	addition
		|	$e_1 - e_2$	subtraction
		|	$e_1 * e_2$	multiplication
		|	[]	nil
		|	$e_1{:}e_2$	cons
		|	`head` e	head of list
		|	`tail` e	tail of list

Type Checker

types	τ	::=	*int*	
		|	*boolean*	
		|	$\tau_1 \to \tau_2$	function type
expressions	e	::=	c	constant
		|	x	variable
		|	$\lambda x : \tau.e$	abstraction
		|	$e_1\ e_2$	application
		|	$e_1 + e_2$	addition
		|	$e_1 - e_2$	subtraction
		|	$e_1 * e_2$	multiplication
		|	e_1 `and` e_2	logical and
		|	e_1 `or` e_2	logical or
		|	`not` e	logical not
		|	$e_1 = e_2$	equality check
		|	$e_1 < e_2$	less than check
		|	$e_1 > e_2$	greater than check
		|	`if` e_1 `then` e_2 `else` e_3	conditional