Informal Proceedings of the 12th International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2026)

Lisbon, Portugal, July 19, 2026

Preface

The 12th International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2026) was held on July 19, 2026, in Lisbon, Portugal, as part of FSCD 2026. WPTE brings together researchers working on program transformations, evaluation, and operationally based programming language semantics, using rewriting methods. This volume collects the extended abstracts of the contributions accepted for presentation. We thank all authors for their submissions and the program committee and reviewers for their feedback. We hope the contributions will foster further research and collaboration in the rewriting community.

Invited Talks

- **Staging for Synthesis and Analysis**
  by *Nada Amin*
- **Fully Abstract Normal Form Bisimulation for Call-by-Value PCF**
  by *Nikos Tzevelekos*

Program Committee

- Martin Avanzini, Inria Sophia Antipolis
- Carsten Fuhs (co-chair), Birkbeck, University of London
- Jan-Christoph Kassing, RWTH Aachen University
- Thomas Kœhler, ICube Lab, CNRS, Université de Strasbourg
- Misaki Kojima, Nagoya University
- Rubén Rubio, Universidad Complutense de Madrid
- Traian Şerbănuţă, University of Bucharest
- Germán Vidal, Universitat Politècnica de València
- Janis Voigtländer (co-chair), University of Duisburg-Essen

Contents

An Interactive Proof Mode for Dafny Based on Back Translation of Verification Obligations

Ștefan Ciobâcă
Alexandru Ioan Cuza University
Iași, Romania
stefan.ciobaca@uaic.ro

K. Rustan M. Leino
Amazon Web Services
Seattle, WA, US
leino@acm.org

Ștefan-Alexandru Mercaș
Alexandru Ioan Cuza University
Iași, Romania
smercas@gmail.com

Roxana-Mihaela Timon
Alexandru Ioan Cuza University
Iași, Romania
timonroxana91@gmail.com

We extend the Dafny system with an interactive proof mode. We present how the interactive proof mode can be used on a motivating example, we discuss the underlying architecture of the system, and provide a prototype implementation.

1 Introduction

Dafny [9] is a programming language that supports auto-active verification, a combination of **auto**matic proofs based on SMT automation and inter**active** proofs based on hints provided by the user. Dafny works by translation into the intermediate verification language Boogie [2], which in turn discharges verification obligations using an SMT solver, typically Z3 [11].

Due to the inherent computational complexity of the underlying problem, proof obligations are in general not discharged automatically. Instead, the programmer must guide the prover by providing helper proof annotations. These annotations could consist of helper assertions, loop invariants, read-/write frames, helper lemmas, ghost state/code, and others. Once a proof annotation is added to the code, the entire development needs to be recompiled in order to verify whether the annotation helps the solver.

We propose an Interactive Proof Mode (IPM) for Dafny. The IPM starts with a proof obligation that cannot be discharged automatically by the Z3 solver. It prints the proof obligation as a Boolean-valued Dafny expression that represent the hypotheses and the goal to be proven from the hypotheses. It accepts input from the user in the form of tactics to be applied on the current proof obligation, similar to proof assistants, such as Lean [1] and Rocq [14]. Once the proof obligation is discharged, the IPM prints out Dafny code that represents a formal proof of the original verification condition. The proof can be copied back into the Dafny project of the user at the point where the initial proof obligation occurs in the code and serves as an explanation of why the original assertion that could not be discharged automatically holds. The code now has enough proof annotations for the proof obligation to discharge successfully (unless there is a bug in the IPM).

In Section 2 we discuss a motivating example: we show how a lemma can be proven in stock Dafny, and how our proof mode improves on this process and enables to discharge the lemma interactively. In Section 3 we discuss the architecture of our system. The system consists of two core components that interact with each other: • A branch of the Dafny compiler, which we call the *Dafny Incremental Proof Server* (DIPS), which allows to incrementally add proof hints to a Dafny project and retrieve the resulting verification obligations (discussed in Section 3.1), and • A back translation component based

on rewrite rules, which takes as input SMT-LIB verification obligations generated by the Dafny-Boogie-Z3 pipeline, and presents these verification obligations to the user as Dafny expressions (discussed in Section 3.2).

We discuss related work in Section 4 and we conclude in Section 5. We have presented an earlier version of this paper at the Dafny 2026 workshop (without formal proceedings).

2 Motivating Example

We consider the Dafny lemma in Figure 1, which cannot be proven automatically by the system.

```
lemma triangle_sum_even(x: int)
  ensures x * (x + 1) % 2 == 0
{ assume false; }
```

Figure 1: A Dafny lemma that fails to verify automatically, which we use as a running example.

Because the lemma uses nonlinear integer arithmetic, which is undecidable in general, the SMT solver cannot automatically prove the resulting verification obligation, and proof hints must be provided by the user. When confronted with such a verification obligation, which fails, a Dafny programmer will typically start with the placeholder proof `assume false;`, and enter an iterative refinement process to build a proof of the failing proof obligation. Such a possible flow is shown in Figure 2.

```
lemma triangle_sum_even
  ...
{
 assume false;
}
```

(a) Start with placeholder proof.

```
lemma triangle_sum_even
  ...
{
 assert x * (x + 1) % 2 ==
   (x % 2) * ((x + 1) % 2);
 assume false;
}
```

(b) Helper assertion does not verify.

```
lemma triangle_sum_even
  ...
{
 if (x % 2 == 0) {
  assume false;
 } else {
  assume false;
 }
}
```

(c) Try a case distinction.

```
lemma triangle_sum_even
  ...
{
 if (x % 2 == 0) {
  assert x == 2 * (x / 2);
  assume false;
 } else {
  assume false;
 }
}
```

(d) Helper assertion not sufficient.

```
lemma triangle_sum_even
  ...
{
 if (x % 2 == 0) {
  assert x == 2 * (x / 2);
  assert x * (x + 1) ==
  2 * ((x / 2) * (x + 1));
 } else {
  assume false;
 }
}
```

(e) First branch works.

```
lemma triangle_sum_even
 ...
{
 if (x % 2 == 0) {
  ...
 } else {
  assert x == 2 * (x/2) + 1;
  assert x * (x + 1) ==
  2 * (x * ((x / 2) + 1));
 }
}
```

(f) Second branch is analogous.

Figure 2: The stages that a Dafny programmer could go through to prove the lemma in Figure 1.

This iterative proof development loop described above suffers from at least two issues: • it can be time-consuming (for large code bases) to recompile the entire project with each iteration of the proof, and • the facts known to the prover at the point where the proof obligation fails are not all locally available, in general, but they can be spread throughout the code (e.g., in some loop invariant, in some class invariant, as a well-formedness condition).

Our interactive proof mode (`IPM`) addresses these two issues, by allowing developers to debug in an interactive environment a proof obligation that fails to verify. To enter the interactive proof mode, we add the annotation `{:ipm}` to the failing proof obligation:

```
lemma triangle_sum_even(x: int)
  ensures {:ipm} x * (x + 1) % 2 == 0
{ }
```

and we call a Python script on the `Dafny` source file: `python3 dafny-ipm.py file.dfy`. We choose this mode of interaction as a quick prototype. A more seamless user experience would integrate the IPM in Visual Studio Code, the most popular IDE for Dafny.

The interactive environment resembles that of the `Rocq` [14] or `Lean` [1] provers. The proof context (facts known to the system) and the proof goal (what must be shown) are shown at each iteration:

```
hypotheses: -
goal:       (((x * (x + 1)) % 2) == 0)
```

The system reads a series of proof tactics that help discharge the current proof obligation. For example, the tactic `check`[1] can be used to check whether `Z3` can prove some fact from the current set of hypotheses:

```
> check (x * (x + 1) % 2 == (x % 2) * ((x + 1) % 2))
Not provable.
```

The user receives immediate feedback after each tactic application, since there is no need to recompile the entire code base. The significant sources of lag are calls to the solver (`Z3`), which cannot be avoided, but the timeout can be set to a low value, like 1 second.[2] To solve the goal in our running example above, the user can perform a case analysis using the `case` tactic:[3]

```
> case ((x % 2) == 0)
hypotheses:   ((x % 2) == 0)
goal:         (((x * (x + 1)) % 2) == 0)
[goal 2 is: (((x * (x + 1)) % 2) == 0)]
```

The case analysis tactic above splits the current proof obligation into two cases, depending on the truth value of the expression `((x % 2) == 0)`. For the two cases in our example, the user can provide the proof using the tactic `assert`, which adds a (provable) expression to the current set of hypotheses. After applying a series of tactics that discharge the proof obligation, the IPM prints out a proof of the verification condition:

[1]The `check` tactic does not directly correspond to a `Dafny` construction. Instead, we implement it as an assertion, using the `assert` command. After the verification obligation is queried, the assertion is forgotten by IPM and is not used in subsequent calls. We choose the name `check` because there is a similar command in the `B3` intermediate verification language (`https://github.com/dafny-lang/b3`).

[2]Setting a larger timeout in the interactive proof mode can lead to a proof obligation discharging automatically in the IPM (due to a larger timeout), but not in stock `Dafny` compilation pipeline (due to a smaller timeout).

[3]The `case` tactic creates in the background an `if _ { _ } else { _ }` statement. For the tactic name, we prefer to use `case` instead of `it`, because the underlying proof rule is *case analysis*.

```
if (((x % 2) == 0)) {
  assert (x == (2 * (x / 2)));
  assert ((x * (x + 1)) == (2 * ((x / 2) * (x + 1))));
} else {
  assert (x == ((2 * (x / 2)) + 1));
  assert ((x * (x + 1)) == (2 * (x * ((x / 2) + 1))));
}
```

The proof code can be copied back into the Dafny project of the user, which will then verify successfully.

3 How it works

The diagram in Figure 3 shows the architecture of the interactive proof mode.

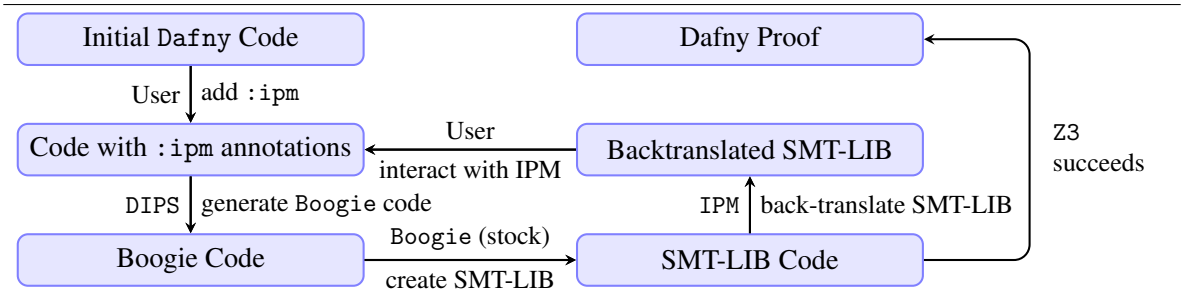


Figure 3: IPM Architecture

After the user annotates the proof obligation for which the IPM should be started with the {:ipm} attribute, the system starts the Dafny Incremental Proof Server (DIPS) on the initial source code. The server runs the stock Dafny-Boogie-Z3 compilation pipeline on an instrumented version of the code and generates SMT-LIB code. The SMT-LIB encoding of the verification condition can be difficult to read by a human. Therefore, the IPM back translates it to a boolean-valued Dafny expression using a series of program transformations that we document in this paper. The Dafny expression is presented to the user. When the user calls a tactic, the IPM translates the tactic into Dafny code and queries the DIPS for the updated verification obligation(s). This is repeated in a loop until the initial verification obligation is fully discharged.

3.1 Dafny Incremental Proof Server (DIPS)

To enable the construction of an interactive proof mode, we branch the Dafny compiler to create the *Dafny Incremental Proof Server* (DIPS), which serves two purposes: • **support incremental verification**, by allowing to add proof structure into the original Dafny project and generate the updated verification conditions; and • **add instrumentation to the Dafny code** in order to enable the back translation component. We explain how the DIPS works on our running example:

```
lemma {:isolate_assertions} {:induction false} m(x: int)
  ensures {:ipm} x * (x + 1) % 2 == 0
{
}
```

The {:isolate_assertions} option is used to force the verifier to generate one verification obligation per assertion, instead of potentially merging several assertions into one verification obligations. The IPM that we propose works best when {:isolate_assertions} is used. The {:induction false} option stops the automated induction heuristic. The IPM works just as well, regardless of whether {:induction false} is present or not, but in the absence of {:induction false} there is an additional hypothesis (the induction hypothesis) which is not helpful in our example. When run on the file above, the IPM starts the DIPS, which first transforms the code as follows:

1. it creates a module _protectors, which contains helper functions that are needed in the back translation process, which we explain in more detail later, in Section 3.2;
2. it moves the entirety of the original source code into a module called _IPM;
3. for each entity (lemma, function, method) that contains an {:ipm} annotation, it creates one module (in our example, _IPM_m), refining the _IPM module above and containing a copy of the entity.

For our running example, the resulting code is the following (edited for brevity):

```
module _protectors {
  function _protect<T>(x: T, name: string): bool { true }
  function _protectScope<T>(x: T, name: string): bool { true }
  function _protectToProve(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
  function _protectToProveImmediate(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
  function _protectToProveWF(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
  function _protectToProveInv(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
  function _protectFinishedInv(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
}
abstract module _IPM {
  lemma {:isolate_assertions} {:induction false} m(x: int)
    requires _protectors._protect(x, "x")
    ensures x * (x + 1) % 2 == 0
    decreases x
  { }
  import _protectors
}
abstract module _IPM_m refines _IPM {
  lemma {:isolate_assertions} {:induction false} _IPM_m(x: int)
    requires _protectors._protect(x, "x")
    ensures {:ipm} _protectors._protectToProve(x * (x + 1) % 2 == 0,
      "x * (x + 1) % 2 == 0", [_protectors._protectScope(x, "x")], 0)
    decreases x
  { }
}
```

The DIPS then runs the stock Dafny compilation pipeline to create the verification conditions. For the example above, it generates the following verification condition, expressed in SMT-LIB format (some parts elided for brevity):

```
(assert (not (=> (= (ControlFlow 0 0) 2) (let ((anon0_correct (=> (and
 (and (and (and ($IsGoodHeap $Heap@@2) ($IsHeapAnchor $Heap@@2))
 (|__protectors.__default.__protect#canCall| TInt ($Box_577 |x#0@@32|)
 [...])) (and (__protectors.__default.__protect TInt
 reveal__protectors._default._protect ($Box_577 |x#0@@32|) (Lit_29427
 (|Seq#Build| |Seq#Empty| ($Box_28869 (|char#FromInt| 120))))) (=
 $_ModifiesFrame@0 (|lambda#0| null $Heap@@2 alloc false)))) (and (and
```

```
(|__protectors.__default.__protectScope#canCall| TInt ($Box_577
|x#0@@32|) [...])  (|__protectors.__default.__protectToProve#canCall|
TBool ($Box_1061 (= (Mod (Mul |x#0@@32| (+ |x#0@@32| 1)) (LitInt 2))
(LitInt 0))) [...] [...] (LitInt 0))) (= (ControlFlow 0 2) (- 0 1))))
($Unbox_1061 (__protectors.__default.__protectToProve TBool
reveal__protectors._default._protectToProve ($Box_1061 (= (Mod (Mul
|x#0@@32| (+ |x#0@@32| 1)) (LitInt 2)) (LitInt 0))) [...]  [...]
(LitInt 0)))))) anon0_correct)) ))
```

The core part in the verification obligation is on the last few lines:

```
(= (Mod (Mul |x#0@@32| (+ |x#0@@32| 1)) (LitInt 2)) (LitInt 0))
```

As you can see, the SMT-LIB code is not readable, but it contains all information needed by our back translation component to pretty print it as a user friendly `Dafny` expression.

The IPM then reads a tactic from the user and translates the tactic into a partial `Dafny` proof, which is sent to the DIPS. For example, if the user starts with a case analysis on `x % 2 == 0`, the DIPS will receive a command similar to
`0ph:  if (x % 2 == 0) { } else { }`,
which means to insert as a **p**roof **h**int for the verification obligation with an index of **0** (hence `0ph:`) the text `if (x % 2 == 0) { } else { }`.

The DIPS will then insert the text into the source code at the right place in the module corresponding to the current lemma:

```
abstract module _IPM_m refines _IPM {
  lemma {:isolate_assertions} {:induction false} _IPM_m(x: int)
    requires _protectors._protect(x, "x")
    ensures {:ipm} _protectors._protectToProve(x * (x + 1) % 2 == 0,
      "x * (x + 1) % 2 == 0",
      [_protectors._protectScope(x, "x")], 0)
    decreases x
  {
    if x % 2 == 0 {
    } else {
    }
  }
```

The server will then run the stock `Dafny` pipeline and regenerate the verification conditions. We modify the copy of the lemma living in its own module for efficiency. The more expensive phases of the pipeline are cached from the initial compilation for each module separately. These costly phases are therefore only run for the modified module, which is typically small. Therefore the regeneration of the verification conditions is much faster than if the entire source code were compiled from scratch. To perform a preliminary evaluation of the effectiveness of the incremental resolver, we profiled the resolution phase both with and without result reuse enabled using the standard library of `Dafny` as the benchmark. When we perform resolution from scratch, we find an average execution time of approximately 4.49s on the `Dafny` standard library files in `Source/DafnyStandardLibraries/src/Std` on a typical laptop. With reuse enabled, the same process completed in an average of approximately 0.14s, corresponding to a speedup of about 32× **for the resolution phase only**. For a benchmark consisting of three files in the standard library (`Base64.dfy`, `BoundedInts.dfy`, `Wrappers.dfy`), we measured 6.4s for the stock compilation pipeline. Adding one assertion programatically and generating the corresponding VCs using DIPS takes about 0.12s, which is fast enough for the IPM.

3.2 Back Translation of Verification Obligations

As discussed above, the back translation engine takes as input the SMT-LIB code produced by the Dafny-Boogie-Z3 pipeline and pretty prints the verification obligations as `Dafny` expressions. The IPM uses the SMT-LIB parser provided in the Z3 bindings for Python to read the solver configuration options (omitted for brevity), around 1000 axioms, and then the verification obligations.

For the lemma in the running example (Figure 1), we show in Figure 4 the SMT-LIB string generated by stock `Dafny` (not DIPS):

```
[...]
(assert (forall ((o@@5 T@Box) ) (! (not (|Set#IsMember| |Set#Empty| o@@5))
 :qid |filebpl.796:15| :skolemid |125| :pattern ( (|Set#IsMember| |Set#Empty| o@@5))
)))
[...]
(push 1)
[...]
(assert (not (=> (= (ControlFlow 0 0) 2) (let ((anon0_correct (=> (and (and
($IsGoodHeap $Heap) ($IsHeapAnchor $Heap)) (and (= $_ModifiesFrame@0  (|lambda#0|
null $Heap alloc false)) (= (ControlFlow  0 2) (- 0 1)))) (= (Mod (Mul |x#0@@1|
(+ |x#0@@1| 1)) (LitInt 2)) (LitInt 0))))) anon0_correct)) ))
(check-sat)
[...]
(pop 1)
```

Figure 4: SMT-LIB verification condition for the running example. The core of the proof goal is `(= (Mod (Mul |x#0@@1| (+ |x#0@@1| 1)) (LitInt 2)) (LitInt 0))`, which corresponds to the `Dafny` postcondition $x*(x+1)\%2 = 0$.

The verification obligation shown in Figure 4 exhibits key difficulties that must be handled during back-translation:

i) the identifier `x` in the initial source code is now mangled: it appears as `x#0@@1` in SMT-LIB;

ii) the arithmetic operations are symbols like `Mod` and `Mul`, which are axiomatized in the SMT-LIB code;

iii) the constants are boxed (e.g., `LitInt 2` is used for the constant 2), and

iv) additional constructions, like `$IsGoodHeap`, are not present in the original source code, but are an artifact of the translation process.

In order to be able to back translate the verification obligations, we start not with the SMT-LIB code generated by stock `Dafny`, but with the SMT-LIB file generated by the `DIPS`, which contains some helper annotations that we discuss next. Recall the instrumented `Dafny` code for our running example:

```
module _protectors {
  function _protect<T>(x: T, name: string): bool { true }
  function _protectScope<T>(x: T, name: string): bool { true }
  function _protectToProve(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
  function _protectToProveImmediate(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
  function _protectToProveWF(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
  function _protectToProveInv(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
  function _protectFinishedInv(x: bool, name: string, scope: seq<bool>, id: int): bool { x }
```

```
}
abstract module _IPM_m refines _IPM {
  lemma {:isolate_assertions} {:induction false} _IPM_m(x: int)
    requires _protectors._protect(x, "x")
    ensures {:ipm} _protectors._protectToProve(x * (x + 1) % 2 == 0,
      "x * (x + 1) % 2 == 0", [_protectors._protectScope(x, "x")], 0)
    decreases x
  { }
}
```

We use names starting with _ (underscore) because they are not accepted by the `Dafny` parser as valid identifiers. Therefore, they cannot clash with user-defined identifiers and effectively add as reserved keywords. We create these names programmatically directly in the abstract syntax tree, after the parsing phase. These names are used successfully as identifiers in subsequent compilation passes. We use the `_protect*` family of functions to box certain expressions, in order to provide helper information needed for back translation: • `_protect`**:** is used to map each program variable with its `Dafny` name, so that mangled identifiers occurring in the SMT-LIB code (such as `|x#0@@1|`) can be mapped back in a principled manner to their original name (such as `x`); • `_protectScope`**:** is used to disambiguate variables in the presence of name shadowing; • `_protectToProve`**:** is used to tag the verification obligations that should go into the IPM because they have the {`:ipm`} attribute; • `_protectToProveImmediate`**:** is used to tag the temporary verification obligations that are generated by the Interactive Proof Mode during the interactive proof creation process, which we call *immediate* proof obligations. • `_protectToProveWF`**:** is used to tag the well-formedness conditions generated from a protected proof obligation, preserving their association with the original proof obligation; • `_protectToProveInv`**:** is used to tag the verification obligations corresponding to loop invariants, allowing the Interactive Proof Mode to distinguish them from non-invariant proof obligations; • `_protectFinishedInv`**:** is used to mark invariant proof obligations whose corresponding loops have already finished, allowing the Interactive Proof Mode to distinguish them from currently active invariants;

The back translation engine then performs a number of passes over the SMT-LIB expressions to transform them into readable `Dafny` syntax.

Phase 1: removal of `_protect*` annotations We remove all wrapper calls to the `_protect` function. During this removal, we simultaneously extract and maintain a mapping for the correspondence between the initial `Dafny` identifiers and their mangled SMT-LIB counterparts.

Example: The protection wrapper `_protectors._protect(x, "x")` could occur in the SMT-LIB code as

```
(__protectors.__default.__protect TInt reveal__protectors._default._protect
  ($Box_577 |x#0@@32|)
  (Lit_29427 (|Seq#Build| |Seq#Empty| ($Box_28869 (|char#FromInt| 120)))))
```

Such calls to `_protect` are removed, and the mapping between the SMT-LIB identifier `|x#0@@32|` and the `Dafny` identifier `x` (character code 120) is stored as a side effect.

Phase 2: pattern replacement We replace axiomatized SMT-LIB symbols with their `Dafny` counterparts, as exemplified in Table 1.

For example, expressions such as `(|char#FromInt| 120)`, `(LitInt 2)`, `(Mod (Mul (LitInt 2) (LitInt 3) (LitInt 4)))` are rewritten into: `'x'`, 2, and `2 * 3 % 4`, respectively.

SMT-LIB Pattern	Dafny Output	Use Case
`LitInt n`	`n`	Integer constants
`|char#FromInt| n`	character with Unicode code `n`	Character literals
`Seq#Length s`	`|s|`	Sequence length
`Seq#Index seq i`	`seq[i]`	Sequence element access
`Set#Union s1 s2`	`(s1 + s2)`	Set union
`MultiSet#Multiplicity ms e`	`ms[e]`	Element multiplicity
`Map#Build m k v`	`m[k := v]`	Map update
`Mod a b`	`(a % b)`	Modulo
`= a b`	`a == b`	Equality
`not (= x y)`	`x != y`	Inequality
`not (e in s)`	`(e !in s)`	Negated membership
`or true x`	`true`	ssimplification
`=> true x`	`x`	simplification
`Constructor args`	`Constructor(args)`	Constructor call
`Destructor obj`	`obj.field`	ADT field access
other	...	...

Table 1: Example of rewrite rules used to present the verification obligation as `Dafny` expressions.

SMT-LIB pattern to remove	Reason
`ControlFlow x y`	Used only for error reporting
`$IsHeapAnchor h`	Implicit in the `Dafny` code
`$IsGoodHeap h`	Implicit in the `Dafny` code
`= $_ModifiesFrame0 f`	Frame annotations not supported in IPM yet
`$IsAlloc* x t h`	Heap allocation not supported in IPM yet

Table 2: Patterns which are removed from the verification obligation.

Phase 3: variable names (simultaneous with Phase 2) We replace SMT-LIB variables names, such as `|x#0@@32|`, with the `Dafny` counter-part computed during Phase 1. During this phase, we also handle name shadowing by prioritizing the names occurring in the `_protectScope` function.

Phase 4: removal of implicit verification predicates We eliminate all calls to symbols that are related to error reporting, to predicates that are implicit at the `Dafny` level, and to parts of the language that we do not yet support, as summarized in Table 2.

Phase 5: selection of tagged verification obligations We identify which of the SMT-LIB verification obligations should be presented to the user by examining the SMT-LIB goal for the protection functions `_protectToProve` and `_protectToProveImmediate` (and we remove the calls to these functions). Verification obligations tagged by `_protectToProve` are generated by {`:ipm`} annotations in the initial source code. The verification obligations tagged by `_protectToProveImmediate` are created for the holes in the current partial proof being created using the IPM.

Phase 6: split into hypotheses and goal If, after normalization, the verification obligation is of the form $H_1 \wedge H_2 \wedge H_n \rightarrow G$, we present H_i ($1 \leq i \leq n$) as the hypotheses to the current verification obligation and G as its goal.

User interaction The hypotheses and the goal are shown to the user, who should apply tactics until the goal is discharged successfully.

4 Related Work

We relate our work to two large categories of verification systems: verifiers that rely on SMT solvers (like Why3 [5] and F* [7]), and verifiers based on dependent type theory or higher-order logic (like Rocq [17], Lean [1], or Isabelle/HOL [12]). Most auto-active verification systems [5, 7, 13, 10] rely on helper assertions to guide the underlying solver, while type theory based systems typically rely on an interactive proof mode.[4] Our work can be seen as the beginning of a bridge extending from the first approach towards the second. Out of the SMT-solver based approaches, the Why3 deductive verifier [5] typically relies on annotations (invariants, assertions), but also supports for interactive proofs. The interactive mode of Why3 is much more mature than our prototype for `Dafny`. However, unlike the proof mode that we propose, the proof object that is obtained in Why3 using the interactive mode is hidden, and cannot be placed into the initial source code to serve as an explanation of why the verification obligation holds. Another approach to the same problem of integrating automated and interactive provers is to add automation tactics into interactive theorem provers [4]. Recent work [18] shows how to combine interactive and automated proofs for `C` programs in the `Rocq` prover. A monadic shallow embedding of an executable program semantics into `Lean` is used to obtain a framework [8] where `Dafny`-style specifications can be discharged using a combination of off-the-shelf SMT solvers and interactive proofs. An embedding [15] of a fragment of `Dafny` into `Lean` with the goal of allowing for interactive proofs was presented at the `Dafny` 2025 workshop. The separation logic Iris framework was recently enhanced with heap-dependent assertions [16]. The `Z3` Axiom Profiler [3] can be used to understand why certain proof obligations involving quantifiers are slow to verify. Chakarov et al. propose a method [6] to back-translate SMT counterexamples to `Dafny` syntax. `Knuckledragger` [19] is an effort to turn `Z3` into an interactive theorem prover. We have presented an earlier version our IPM at the `Dafny` 2026 workshop (without formal proceedings). The main difference is that the earlier version did not rely on the DIPS. Instead, the tactics processed verification obligations at the Python level, therefore being potentially less faithful to the stock verifier than the present version. Other differences include various improvements and supporting a larger subset of Dafny.

5 Discussion

Our current implementation is a prototype that works for a significant subset of `Dafny`. The IPM, with instructions on running it, is available at `https://github.com/smercas/docker-for-conf`. It has some limitations, which we discuss briefly in this section.

The main goal of the proof mode that we propose is to be as faithful as possible to the `Dafny` verification chain, without re-implementing the verification chain itself in the interactive proof mode. In order to ensure faithfulness, the IPM calls the `Z3` solver with the same configuration options that `Dafny`

[4] F* is different, because it is based on type theory, but relies on SMT automation.

uses: we use it in incremental mode, we turn off the model-based quantifier instantiation option, etc. We currently concentrate on proof obligations generated with the `{:isolate-assertions}` option and we do not support proof obligations arising from well-formedness conditions.[5] The DIPS code is a fork of the `Dafny` system, but in the future we plan to integrate it back into stock `Dafny` as a command line option.

The proof mode currently supports only a restricted subset of `Dafny`, including Booleans, integers, arithmetic operations, functions, ADTs, and sets. In future work, we plan to extend this subset, continuing with the imperative and object-oriented features. The prototype supports a restricted set of tactics: `check`, `assert`, `assume`, `case`. These tactics correspond very closely to `Dafny` code constructions. We plan to expand the set of tactics in future work to account for more complex reasoning steps, such as induction. We currently rely on the Python bindings for `Z3` to parse the SMT-LIB code output by Boogie and we currently only support a limited set of meta-commands, such as `quit`, `search`, `set-timeout`. A difficulty that we foresee in future work is to back-translate VCs for imperative programs, which can mention a particular program variable at various points in time. There is no `Dafny` syntax to refer to a previous value of a variable (unless saved into a ghost field), and therefore the IPM will have to use some new notation for this case.

Another difficulty that we foresee is the back-translation of more complicated expressions. Consider a `Dafny` declaration `var x: C`, where `C` is a class. Such a variable declaration is translated into a conjunction, `n != null && dtype(x) == C`. We plan for the proof mode to identify these patterns and present them in `Dafny` syntax to the user. The protection functions that we use could interfere with trigger generation, and we will update the trigger generation algorithm to treat calls to `_protect*` as being invisible.

The main advantages of using the IPM over manually writing the proof directly in `Dafny` are: • the full context is shown to the user at any given point, including constraints that the compiler inserts and which are not visible at all in the source code, and • the user can experiment with the current proof obligation without going through the potentially time consuming edit-compile-verify cycle. As future work, we could also experiment with showing provenance information: where in the source code each part of the hypotheses/goal originates, which could help the user further understand the verification obligation.

Currently, we focus on the technical aspects of the IPM, but we could also do a user study in the future, in order to understand evaluate the usefulness of the tool. Finally, the IPM that we propose could be extended to modify existing proofs, could be integrated with the `Z3` axiom profiler, and be made available as a plugin to `Dafny` IDEs, like Visual Studio Code. In an IDE plugin, the hypotheses and the goal could be back translated and shown in a separate window, and tactic applications could directly modify the source code.

We have validated the IPM framework empirically against a suite of 50 `Dafny` programs that cover the subset of the language that we currently support: • multi-set operation variants (empty, membership, cardinality, union, intersection, difference, etc.); • map operation tests (construction, lookup, update, projections, etc.); • infinite set operation tests; • finite set operation tests; • function recursion patterns and fuel handling; • arithmetic and logical expressions; • ADTs (destructors, constructors, discriminators).

All examples have been processed using the Dafny-Boogie-Z3 pipeline and validated with the IPM back-translation engine to ensure correct transformation of verification conditions into human-readable `Dafny` syntax.

[5] We did have well-formedness conditions working in an earlier prototype of the IPM, which did not make use of the DIPS.

Remarks

We have used AI to help typeset early versions of the TikZ diagrams. The article itself was fully written by us, with no AI assistance. We've used AI to generate the IPM's source code, for example, to set up a testing framework or to speed up writing recursive functions over Z3 expressions. The core algorithms (e.g., the back-translation algorithm) were all designed by us.

References

[1] Jeremy Avigad, Gabriel Ebner & Sebastian Ullrich (2017): *The Lean Reference Manual*. Available at `https://leanprover.github.io/reference/`. Accessed: 2025-10-01.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs & K. Rustan M. Leino (2006): *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf & Willem-Paul de Roever, editors: *Formal Methods for Components and Objects*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 364–387.

[3] Nils Becker, Peter Müller & Alexander J. Summers (2019): *The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations*. In Tomáš Vojnar & Lijun Zhang, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 99–116.

[4] Jasmin Blanchette, Mathias Desharnais, Lawrence C. Paulson & Lukas Bartl (2025): *Hammering Away: A User's Guide to Sledgehammer for Isabelle/HOL*. Institut für Informatik, Technische Universität München. Available at `https://isabelle.in.tum.de/dist/doc/sledgehammer.pdf`. Accessed: 2025-10-01.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond & Andrei Paskevich (2025): *The Why3 Platform — Documentation*. Available at `https://www.why3.org/doc/`. Version 1.8, accessed: 2025-10-01.

[6] Aleksandar Chakarov, Aleksandr Fedchin, Zvonimir Rakamarić & Neha Rungta (2022): *Better Counterexamples for Dafny*. In Dana Fisman & Grigore Rosu, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 404–411.

[7] F* Development Team (2025): *F* Reference Manual / Documentation*. Available at `https://fstar-lang.org/`. Accessed: 2025-10-01.

[8] Vladimir Gladshtein, George Pîrlea, Qiyuan Zhao, Vitaly Kurin & Ilya Sergey (2026): *Foundational Multi-Modal Program Verifiers*. *Proceedings of the ACM on Programming Languages* 10(POPL). Available at `https://verse-lab.github.io/papers/loom-preprint.pdf`. To appear.

[9] K. Rustan M. Leino (2010): *Dafny: An Automatic Program Verifier for Functional Correctness*. In Edmund M. Clarke & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 348–370.

[10] John W. McCormick & Peter C. Chapin (2015): *Building High Integrity Applications with SPARK*. Cambridge University Press, Cambridge, United Kingdom.

[11] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340.

[12] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9. Available at `https://doi.org/10.1007/3-540-45949-9`.

[13] David J. Pearce, Mark Utting & Lindsay Groves (2022): *Verifying Whiley Programs with Boogie*. *J. Autom. Reason.* 66(4), p. 747–803, doi:10.1007/s10817-022-09619-1. Available at `https://doi.org/10.1007/s10817-022-09619-1`.

[14] Rocq Prover Team (2025): *Rocq Prover Reference Manual*. Available at `https://rocq-prover.org/doc/master/refman/index.html`. Accessed: 2025-10-01.

[15] Wojciech Różowski, Georges-Axel Jaloyan & Sean McLaughlin (2025): *Lean on Dafny: Exploring Interactive Verification of Dafny Programs in Lean*. Talk presented at Dafny 2025. Session: Backends and Teaching, chaired by Stefan Zetzsche. Time: 14:18–14:36.

[16] Simon Spies, Niklas Mück, Haoyi Zeng, Michael Sammler, Andrea Lattuada, Peter Müller & Derek Dreyer (2025): *Destabilizing Iris*. *Proc. ACM Program. Lang.* 9(PLDI), doi:10.1145/3729284. Available at `https://doi.org/10.1145/3729284`.

[17] The Coq Development Team (2017): *The Coq Proof Assistant: Reference Manual, version 8.7.1*. Available at `https://coq.inria.fr/doc/V8.7.2/refman/index.html`. Accessed: 2025-10-01.

[18] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel & Qinxiang Cao (2024): *VST-A: A Foundationally Sound Annotation Verifier*. *Proc. ACM Program. Lang.* 8(POPL), doi:10.1145/3632911. Available at `https://doi.org/10.1145/3632911`.

[19] Philip Zucker (2025): *Knuckledragger: A Low Barrier Proof Assistant*. Available at `https://github.com/philzook58/knuckledragger`. Accessed: 2025-10-01.

Improvement Theory for Probabilistic Call-by-Need

David Sabel
Hochschule RheinMain – University of Applied Sciences and Arts
david.sabel@hs-rm.de

Manfred Schmidt-Schauß
Goethe-University Frankfurt
manfredschauss@gmail.com

This work investigates resource improvements for probabilistic lazy PCF, a simply typed, higher-order call-by-need functional language with natural numbers, branching, fixpoints, and probabilistic choice. Building on deterministic improvement theory, a contextual cost improvement preorder is defined, requiring semantic equivalence and non-increasing expected reduction steps in all contexts. To avoid context quantification, distribution-cost improvement is introduced as an intrinsic preorder that compares per-value conditional expected steps. The main result establishes that both preorders coincide for closed programs of number type. This extends the distribution-equivalence characterisation to the resource-sensitive setting and enables local, context-free improvement proofs. As applications, garbage collection and a surface unique-copying transformation are verified as cost improvements, while deterministic common subexpression elimination is conjectured to exhibit the same property.

1 Introduction

Motivation and goals Optimising transformations are a cornerstone of compiler technology for functional languages [24, 25]: inlining, common subexpression elimination, and let-floating should preserve program meaning and reduce resource usage, such as runtime or memory. The improvement theory of Sands [22, 30] provides the formal foundation: an expression t improves s if both expressions are semantically equivalent and, in every program context C, the cost of $C[t]$ does not exceed that of $C[s]$. Improvement theory for deterministic call-by-need functional languages has been developed extensively [10, 11, 12, 13, 14, 22, 26, 30, 33, 36] (we discuss details below).

Probabilistic programs arise in machine learning, Bayesian inference, randomised algorithms, and simulation [5]. While these programs are important, improvement theory for probabilistic programs is essentially unexplored. The goal of this paper is to start the development of such an improvement theory.

Optimisation of probabilistic programs has to concern *expected* cost: replacing expression s by a cheaper but equivalent expression t should reduce expected cost in every context. In this paper we focus on runtime as the cost measure, counting expected number of reduction steps.

Approach and results We work with $\mathit{ProbPCFR}^{need}$, the probabilistic lazy PCF [28, 34], a typed call-by-need lambda calculus extended by let-expressions, natural numbers, operations on numbers, branching, and a probabilistic choice $s \overset{p}{\oplus} t$ that reduces to s with probability p and to t with probability $1-p$. Expressions are contextually equivalent if they induce the same expected convergence in every program context. For expressions of number type, contextual equivalence has been fully characterised in [28, 34]: closed expressions of number type are contextually equivalent if and only if each output value is produced with the same probability by both expressions; equivalently, they are *distribution equivalent*.

We define a cost semantics for $\mathit{ProbPCFR}^{need}$ counting essential reduction steps. We introduce two improvement preorders: *contextual cost improvement*, which requires semantic equivalence and lower expected cost in every context, and *distribution-cost improvement*, an intrinsic preorder that requires the

same output distribution and, for each output value, no more expected reduction steps conditioned on producing that value. Our main result is that for closed programs of number type, these two preorders coincide. This extends the distribution-equivalence characterisation of contextual equivalence to the resource-sensitive, ordered setting. Finally, we verify transformations, including garbage collection and surface unique copying, as cost improvements, and conjecture that common subexpression elimination for deterministic subexpressions is a cost improvement, explaining the intuition and the remaining gap.

Related work The foundations of improvement theory for deterministic call-by-need calculi are in [22, 30, 33]: a context lemma, the tick algebra, correctness of transformations, and abstract machine semantics. Further directions include shared-work decorations [27, 32], space improvements [10, 31], worker/wrapper transformations [11], polymorphic improvements [13], clairvoyant semantics [14], equational cost reasoning [12], and improvements from free theorems [36]. Improvement theory is extended to a concurrent call-by-need language in [35]. In [15] a different methodology is applied in Liquid Haskell, using refinement types and a tick monad to verify unary and relational resource bounds.

Little prior work combines improvement theory and probabilistic languages. In [7] an effectful improvement theory for a call-by-value lambda calculus is developed and instantiated to probabilistic effects. Our contribution differs in being call-by-need and in giving a distributional characterisation of cost improvement for closed programs of number type. To our knowledge, this is the first improvement theory for a probabilistic call-by-need language.

Contextual equivalence for probabilistic call-by-need is studied in [29] for recursive let, focusing on program transformations, and in [28, 34] for probabilistic lazy PCF, focusing on the coincidence of contextual and distribution equivalence. More broadly, contextual equivalence in higher-order probabilistic languages is studied via probabilistic applicative bisimilarity [8], step-indexed logical relations [6], complete characterisations for languages with continuous random variables [37], and denotational full abstraction for a call-by-name probabilistic PCF [9]. These works address qualitative equivalence, while we extend the distribution-based characterisation to a quantitative, resource-sensitive preorder.

Expected-cost analysis for probabilistic programs is studied from several directions: weakest-precondition calculi for expected runtimes and the distinction between almost-sure, positive, and bounded almost-sure termination, whose decision complexities differ once nondeterminism is admitted [16, 17, 20], type-based and denotational cost analysis for higher-order probabilistic programs [2, 4, 19, 38], expected-cost analysis via continuation-passing transformations [1], and automatic resource analysis for imperative probabilistic programs [3, 23, 39], as well as for probabilistic integer programs and term rewriting [18, 21]. While these works analyse or bound the cost of a single program, our preorder is relational and asks whether one program is cheaper than another in every context. The notions of almost-sure and positive almost-sure termination are conceptually close to our approach: we also study expected runtime, but result-wise rather than only globally. In our purely probabilistic setting, positive and bounded almost-sure termination coincide [20].

Outline Section 2 recalls $\mathit{ProbPCFR}^{need}$, Section 3 defines the cost semantics and the improvement relations. Section 4 proves the main theorem. Section 5 considers transformations. Section 6 concludes.

2 The Language $\mathit{ProbPCFR}^{need}$

We recall the syntax, reduction, and semantics of $\mathit{ProbPCFR}^{need}$ from [28, 34].

Definition 2.1 (Syntax of Expressions and Types)**.** Let *Var* be an infinite countable set of variables. The syntax of *expressions* $s,t \in Expr$ and *types* $\sigma,\tau \in Typ$ is as follows, where $n \in \mathbb{N}$ and $p \in (0,1)$

$$s,t ::= x \mid \lambda x.s \mid (s\ t) \mid s \overset{p}{\oplus} t \mid \texttt{let}\ x{=}s\ \texttt{in}\ t \mid \texttt{if}\ s\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid \texttt{fix}\ s \mid \texttt{pred}\ s \mid \texttt{succ}\ s \mid n$$
$$\sigma,\tau ::= nat \mid \tau \to \sigma$$

The construct $s \overset{p}{\oplus} t$ is a *probabilistic choice* reducing to s with real valued probability $p \in (0,1)$ and to t with probability $1-p$. We write $s \oplus t$ for $s \overset{0.5}{\oplus} t$. For example, $0 \oplus 1$ is a fair choice between 0 and 1.

We omit the details of the monomorphic type system (see [34]). The expression $s \overset{p}{\oplus} t$ is well-typed with type τ iff both s and t have type τ. Binders are λ and `let` (non-recursive). This induces the usual notions of free and bound variables of expression t, written $FV(t)$ and $BV(t)$, as well as open and closed expressions. A closed expression is a *program*, a program of type *nat* is a *stochastic program*.

A context C is an expression with a (typed) hole $[\cdot_\sigma]$ at expression position. We write $C[s]$ for filling the hole of C with expression $s:\sigma$. For instance, $C = \texttt{succ}\ [\cdot_{nat}]$ is a context, and $C[3] = \texttt{succ}\ 3$. The operational semantics uses reduction contexts R, applicative contexts A, flat applicative contexts A^1, and LR-contexts (for *let-right*), which represent an environment of bindings as a stack of `let`-expressions:

$$A ::= [\cdot] \mid (A\ s) \mid \texttt{if}\ A\ \texttt{then}\ s\ \texttt{else}\ t \mid \texttt{pred}\ A \mid \texttt{succ}\ A \mid \texttt{fix}\ A \qquad \mathrm{LR} ::= [\cdot] \mid \texttt{let}\ x{=}s\ \texttt{in}\ \mathrm{LR}$$
$$A^1 ::= ([\cdot]\ s) \mid \texttt{if}\ [\cdot]\ \texttt{then}\ s\ \texttt{else}\ t \mid \texttt{pred}\ [\cdot] \mid \texttt{succ}\ [\cdot] \mid \texttt{fix}\ [\cdot] \qquad R ::= \mathrm{LR}[A] \mid \mathrm{LR}[\texttt{let}\ x{=}A\ \texttt{in}\ R[x]]$$

We will also use *multi-contexts* $C[\cdot_{1,\sigma},\ldots,\cdot_{N,\sigma}] : \tau$. A multi-context is a context with $N \geq 0$ typed holes. Plugging expressions $s_1,\ldots,s_N : \sigma$ into holes $1,\ldots,N$ yields the expression $C[s_1,\ldots,s_N] : \tau$.

Values are naturals $n \in \mathbb{N}$ and abstractions $\lambda x.s$. A *weak head normal form* (WHNF) is an expression of the form $\mathrm{LR}[v]$ with v a value. For a WHNF $\mathrm{LR}[n]$ with $n \in \mathbb{N}$, we write $val(\mathrm{LR}[n]) = n$.

Definition 2.2. *Standard reduction* $\xrightarrow{sr}$ of *ProbPCFR*^{need} is defined by the following rules with $s,t \in Expr$

$(sr,lbeta)$	$R[(\lambda x.s)\ t] \to R[\texttt{let}\ x{=}t\ \texttt{in}\ s]$	$(sr,if\text{-}0)$	$R[\texttt{if}\ 0\ \texttt{then}\ s\ \texttt{else}\ t] \to R[s]$
$(sr,succ)$	$R[\texttt{succ}\ n] \to R[m]$, where $m = n+1$	$(sr,if\text{-}not0)$	$R[\texttt{if}\ n\ \texttt{then}\ s\ \texttt{else}\ t] \to R[t]$, if $n \neq 0$
$(sr,pred)$	$R[\texttt{pred}\ n] \to R[m]$, where $m = \max(0,n-1)$	(sr,cp)	$\mathrm{LR}[\texttt{let}\ x{=}v\ \texttt{in}\ R[x]] \to \mathrm{LR}[\texttt{let}\ x{=}v\ \texttt{in}\ R[v]]$, where v is a value
$(sr,probl)$	$R[s \overset{p}{\oplus} t] \to R[s]$	(sr,fix)	$R[\texttt{fix}\ \lambda x.s] \to R[(\lambda x.s)\ (\texttt{fix}\ \lambda x.s)]$
$(sr,probr)$	$R[s \overset{p}{\oplus} t] \to R[t]$	$(sr,lflata)$	$R[A^1[\texttt{let}\ x{=}s\ \texttt{in}\ t]] \to R[\texttt{let}\ x{=}s\ \texttt{in}\ A^1[t]]$
		$(sr,llet)$	$\mathrm{LR}[\texttt{let}\ x{=}(\texttt{let}\ y{=}s\ \texttt{in}\ t)\ \texttt{in}\ R[x]] \to \mathrm{LR}[\texttt{let}\ y{=}s\ \texttt{in}\ \texttt{let}\ x{=}t\ \texttt{in}\ R[x]]$

We define (sr,lll) as the union of $(sr,llet)$ and $(sr,lflata)$. All rules except $(sr,probl)$ and $(sr,probr)$ are deterministic. Rule $(sr,lbeta)$ is the call-by-need β-reduction. Rule (sr,cp) copies a value to its needed position. We sometimes write $\xrightarrow{sr,a}$ where a denotes the used reduction rule. With $\xrightarrow{sr,+}$ ($\xrightarrow{sr,*}$), we denote the transitive (reflexive-transitive) closure of $\xrightarrow{sr}$, respectively. For example, $(\lambda x.\texttt{succ}\ x)\ 3 \xrightarrow{sr,lbeta} \texttt{let}\ x{=}3\ \texttt{in}\ \texttt{succ}\ x \xrightarrow{sr,cp} \texttt{let}\ x{=}3\ \texttt{in}\ \texttt{succ}\ 3 \xrightarrow{sr,succ} \texttt{let}\ x{=}3\ \texttt{in}\ 4$, which is a WHNF.

We call $\xrightarrow{sr,probl}$ and $\xrightarrow{sr,probr}$ *prob-steps*. A prob-step has a probability weight: if $s \overset{p}{\oplus} t$ is evaluated, then the probability of $\xrightarrow{sr,probl}$ is p, and of $\xrightarrow{sr,probr}$ it is $1-p$. A *prob-sequence* is a finite sequence of pairs (a_i,p_i) with $a_i \in \{probl,probr\}$ and $p_i \in (0,1)$. Given a reduction sequence $s_1 \xrightarrow{sr,a_1} \cdots \xrightarrow{sr,a_n} s_{n+1}$, the corresponding prob-sequence is constructed by adding the probabilities to the prob-steps, only keeping the labels of the prob-steps with the probability weights and removing all other labels. A reduction

Notation	Meaning	Range
$\mathbb{P}(s\downarrow)$	expected convergence of s	$[0,1] \subseteq \mathbb{R}$
$\mathbb{P}(s\downarrow n)$	expected convergence of s with value n	$[0,1] \subseteq \mathbb{R}$
$\mathbb{E}[\mathtt{L}(s)]$	(partial) expected evaluation length of s	$[0,\infty]$
$\mathbb{E}[\mathtt{L}(s); s\downarrow n]$	evaluation length of s to value n	$[0,\infty]$
$\mathbb{E}[\mathtt{L}(s) \mid s\downarrow n]$	conditional evaluation length of s to value n	$[0,\infty]$

Figure 1: Probability and expectation measures of an expression s. Each measure has a *bounded* variant that admits at most k prob-steps, written with a superscript $\leq k$ (e.g. $\mathbb{E}^{\leq k}[\mathtt{L}(s)]$, $\mathbb{P}^{\leq k}(s\downarrow n)$).

sequence $s \xrightarrow{sr,*} t$ is an *evaluation* if t is a WHNF. For an expression s, $Eval(s)$ denotes all prob-sequences of evaluations of s. The probability of a prob-sequence $S = (a_1,p_1),\ldots,(a_n,p_n)$ is $\mathsf{P}(S) = \prod_{(a_i,p_i)\in S} p_i$. Each prob-sequence $S \in Eval(s)$ uniquely determines an evaluation of s (the sequence of reduction steps taken at each prob-choice is fixed by S). We therefore speak of the *evaluation* S of s, using "evaluation" and "prob-sequence" interchangeably. For $s : nat$, we write $S : s \downarrow n$ to mean that $S \in Eval(s)$ and $val(WHNF(s,S)) = n$, i.e. the evaluation S reduces s to $\mathrm{LR}[n]$ with $n \in \mathbb{N}$. We write $|S|$ for the *length* of the prob-sequence S, i.e. if $S = (a_1,p_1),\ldots,(a_k,p_k)$ then $|S| = k$.

As an example, consider $s = (0 \overset{0.3}{\oplus} 1) \oplus 2$. The evaluation $s \xrightarrow{sr,probl} 0 \overset{0.3}{\oplus} 1 \xrightarrow{sr,probr} 1$ has prob-sequence $S = (probl, 0.5), (probr, 0.7)$ with $\mathsf{P}(S) = 0.35$, $|S| = 2$, and $S : s \downarrow 1$. Other prob-sequences in $Eval(s)$ are $(probl, 0.5), (probl, 0.3)$ and $(probr, 0.5)$.

Since the programs are probabilistic, our semantics observes the expected convergence of programs:

Definition 2.3. For a closed expression s, the *expected convergence* is $\mathbb{P}(s\downarrow) := \sum_{S\in Eval(s)} \mathsf{P}(S)$, and the *expected convergence with value* n is $\mathbb{P}(s\downarrow n) := \sum_{S:s\downarrow n} \mathsf{P}(S)$.

For easy reference, an overview of the different probability and expectation measures is in Fig. 1.

Clearly, $\mathbb{P}(s\downarrow) = \sum_{n\in\mathbb{N}} \mathbb{P}(s\downarrow n)$. For $s = 0 \oplus 1$: $\mathbb{P}(s\downarrow 0) = 0.5$, $\mathbb{P}(s\downarrow 1) = 0.5$, $\mathbb{P}(s\downarrow n) = 0$ for $n \geq 2$, and $\mathbb{P}(s\downarrow) = 1$. For $\Omega = \mathtt{fix}\ (\lambda f.f)$: $\mathbb{P}(\Omega\downarrow) = 0$.

Expressions are contextually equivalent if they have the same expected convergence in every context:

Definition 2.4 (Contextual Preorder $\leq_c$ and Equivalence $\sim_c$)**.** Let $s,t : \sigma$. If for all closing contexts $C[\cdot_\sigma] : nat$, $\mathbb{P}(C[s]\downarrow) \leq \mathbb{P}(C[t]\downarrow)$, then $s \leq_c t$. We define $s \sim_c t$ iff $s \leq_c t$ and $t \leq_c s$.

A convenient notion of program equivalence is distribution equivalence, which does not quantify over all contexts but inspects the probability distribution over the natural numbers:

Definition 2.5 (Distribution Approximation $\leq_d$ and Equivalence $\sim_d$)**.** Let s,t be stochastic programs. We write $s \leq_d t$ if for all $n \in \mathbb{N}$: $\mathbb{P}(s\downarrow n) \leq \mathbb{P}(t\downarrow n)$. If $s \leq_d t$ and $t \leq_d s$, then $s \sim_d t$.

We have $0 \oplus 1 \sim_d \mathtt{if}\ (0 \oplus 1)\ \mathtt{then}\ 0\ \mathtt{else}\ 1$, since both produce 0 and 1 each with probability 0.5. However, $0 \oplus 1 \not\sim_d 0 \overset{0.3}{\oplus} 1$, since $\mathbb{P}(0 \oplus 1\downarrow 0) = 0.5 \neq 0.3 = \mathbb{P}(0 \overset{0.3}{\oplus} 1\downarrow 0)$.

The following results in [34] are stated for $p = \frac{1}{2}$, but they also hold for $p \in (0,1)$ as verified in [28]. Hence, we freely use them in their real-valued form, e.g. also for the following result [34, Cor. 4.15]:

Lemma 2.6. For all closed reduction contexts $R[\cdot_{nat}] : nat$ and stochastic programs $s \sim_d t$: $R[s] \sim_d R[t]$.

The main result of [34] is the coincidence of contextual equivalence and distribution equivalence:

Theorem 2.7 ([34])**.** *For stochastic programs s,t: $s \sim_c t \iff s \sim_d t$.*

3 Cost Improvement

We consider the length of evaluations which will be the costs in our improvement definition.

Definition 3.1. For a closed expression s and $S \in Eval(s)$, the *length of evaluation* $\mathtt{L}(s,S)$ is the number of *sr*-steps in the unique evaluation of s with prob-sequence S, *excluding* (sr,lll)-steps and (sr,cp)-steps.

We exclude *lll*-steps and *cp*-steps, since both are administrative: *lll*-steps rearrange the let-binding structure, and *cp*-steps propagate an already-computed value to its use site. In realistic implementations of lazy languages, let-bindings live in a heap: floating them does not cost any runtime work, and accessing a shared binding is an $O(1)$ pointer dereference rather than a genuine computation step. See e.g. [33] for more discussion. In what follows we refer to the steps actually counted by $\mathtt{L}$ as *counted steps*. With $\mathtt{L}(s)$ we denote the random variable for the evaluation length of expression s and define:

Definition 3.2. Let s be a closed expression. The *partial expected evaluation length* is

$$\mathbb{E}[\mathtt{L}(s)] := \sum_{S \in Eval(s)} \mathtt{L}(s,S) \cdot \mathsf{P}(S) \in [0,\infty]$$

The word *partial* reflects that $\mathbb{E}[\mathtt{L}(s)]$ sums only over *terminating* evaluations: diverging paths contribute 0 rather than ∞. This is deliberate, since charging diverging runs cost ∞ would make $\mathbb{E}[\mathtt{L}(s)] = \infty$ if $\mathbb{P}(s{\downarrow}) < 1$, whereas our improvement preorders (Definitions 3.4 and 3.10) relate only semantically equivalent programs, so the divergent probability mass already coincides on both sides ($\mathbb{P}(s{\downarrow}) = \mathbb{P}(t{\downarrow})$, and $\mathbb{P}(s{\downarrow}n) = \mathbb{P}(t{\downarrow}n)$ for all n under $\sim_d$). The partial sum thus isolates the finite cost on which equivalent programs can differ, and non-termination need not be tracked as a separate outcome. However, we use *expected evaluation length* when no confusion can arise.

The value ∞ may occur for programs that converge with positive probability but whose converging paths have unbounded expected length.

Example 3.3. With $add = \mathtt{fix}\ (\lambda f.\lambda x.\lambda y.\mathtt{if}\ x\ \mathtt{then}\ y\ \mathtt{else}\ f\ (\mathtt{pred}\ x)\ (\mathtt{succ}\ y))$, the expression $add\ m\ n$ evaluates to $m+n$ which requires at least m counted steps (one $\mathtt{if}$-expression per recursive call). Define $fun = \mathtt{fix}\ (\lambda h.\lambda n.\ n \oplus (h\ (add\ n\ n)))$ and $s = fun\ 1$. At recursion depth k, the argument is 2^k. With probability $\frac{1}{2}$, we return 2^k, else we pay at least 2^k counted steps and recurse. Hence, $\mathbb{E}[\mathtt{L}(s)] \geq \sum_{k\geq 0} \frac{2^k}{2^{k+1}} = \infty$ while $\mathbb{P}(s{\downarrow}) = 1$.

In the probabilistic-program analysis literature [16, 17], a program t is called *almost surely terminating* (AST) if $\mathbb{P}(t{\downarrow}) = 1$, and *positively almost surely terminating* (PAST) if additionally $\mathbb{E}[\mathtt{L}(t)] < \infty$. The program s from Example 3.3 is AST but not PAST.

We use the cost semantics to define the contextual improvement preorder:

Definition 3.4 (Contextual Cost Improvement)**.** Let $s,t : \sigma$. We write $s \preceq_{ci} t$ (t *improves* s) if:

$$\text{(i) } s \sim_c t\text{, and (ii) for all closing contexts } C[\cdot_\sigma] : nat\text{: } \mathbb{E}[\mathtt{L}(C[s])] \geq \mathbb{E}[\mathtt{L}(C[t])].$$

We write $s \asymp_{ci} t$ if $s \preceq_{ci} t$ and $t \preceq_{ci} s$.

The conditions ensure that t improves s if s and t are semantically equal while the runtime is never worsened when replacing s by t in any stochastic program.

As for contextual equivalence, contextual cost improvement has a universal quantification over all contexts which makes proofs of the improvement property hard. We thus also develop an easier characterisation. Again we consider programs of type *nat* and their induced distribution.

Definition 3.5. For a closed expression $s : nat$ and $n \in \mathbb{N}$, the *expected evaluation length to value n*, $\mathbb{E}[\mathtt{L}(s); s{\downarrow}n] \in [0,\infty]$, and the *conditional evaluation length to value n*, $\mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n] \in [0,\infty]$, are

$$\mathbb{E}[\mathtt{L}(s); s{\downarrow}n] := \sum_{S:\, s{\downarrow}n} \mathtt{L}(s,S) \cdot \mathsf{P}(S) \text{ and } \mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n] := \mathbb{E}[\mathtt{L}(s); s{\downarrow}n] / \mathbb{P}(s{\downarrow}n)$$

where $\mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n] := \infty$ if $\mathbb{P}(s{\downarrow}n) = 0$, and we adopt the convention $0 \cdot \infty = 0$.

The measure $\mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n]$ is the counterpart of $\mathbb{P}(s{\downarrow}n)$ for lengths: $\mathbb{P}(s{\downarrow}n)$ is the probability of producing n, while $\mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n]$ is the expected number of steps for producing n.

Example 3.6 (Difference between $\mathbb{E}[\mathtt{L}(s); s{\downarrow}n]$ and $\mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n]$)**.** Let $s = 0 \oplus 1$. Then $\mathbb{E}[\mathtt{L}(s); s{\downarrow}0] = 1 \cdot 0.5 = 0.5$ and $\mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}0] = 0.5/0.5 = 1$. Now let $t = \mathtt{if}\ (0 \oplus 1)\ \mathtt{then}\ 0\ \mathtt{else}\ 1$ (2 counted steps: one prob-step in the test, one if-step, same distribution). Then $\mathbb{E}[\mathtt{L}(t); t{\downarrow}0] = 2 \cdot 0.5 = 1$ and $\mathbb{E}[\mathtt{L}(t) \,|\, t{\downarrow}0] = 1/0.5 = 2$. Thus $\mathbb{E}[\mathtt{L}(s); s{\downarrow}n]$ combines probability and length into a single weighted sum, while $\mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n]$ is the pure average length on runs producing n.

The following proposition shows how to decompose the expected length. It follows by partitioning $Eval(s)$ by the output value and using $\mathbb{E}[\mathtt{L}(s); s{\downarrow}n] = \mathbb{P}(s{\downarrow}n) \cdot \mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n]$. The latter is true by definition and the convention $0 \cdot \infty = 0$.

Proposition 3.7. For all stochastic programs $s : nat$:

$$\mathbb{E}[\mathtt{L}(s)] = \sum_{n \in \mathbb{N}} \mathbb{E}[\mathtt{L}(s); s{\downarrow}n] = \sum_{n \in \mathbb{N}} \mathbb{P}(s{\downarrow}n) \cdot \mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n]$$

Corollary 3.8. Let s,t be stochastic programs with $s \sim_c t$. If $\mathbb{E}[\mathtt{L}(C[s]) \,|\, C[s]{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(C[t]) \,|\, C[t]{\downarrow}n]$ for all closing contexts $C[\cdot_{nat}] : nat$ and all $n \in \mathbb{N}$, then $s \preceq_{ci} t$.

The following lemma describes how a prob-step affects the per-number measures. Its proof is given in Appendix B (as part (3) of Corollary B.2).

Lemma 3.9. Let $s = R[s_1' \overset{p}{\oplus} s_2']$ be a stochastic program, and let $s_i := R[s_i']$ for $i = 1,2$, i.e. $s \xrightarrow{sr,probl} s_1$ and $s \xrightarrow{sr,probr} s_2$. Then $\mathbb{P}(s{\downarrow}n) = p \cdot \mathbb{P}(s_1{\downarrow}n) + (1-p) \cdot \mathbb{P}(s_2{\downarrow}n)$ as well as $\mathbb{E}[\mathtt{L}(s); s{\downarrow}n] = \mathbb{P}(s{\downarrow}n) + p \cdot \mathbb{E}[\mathtt{L}(s_1); s_1{\downarrow}n] + (1-p) \cdot \mathbb{E}[\mathtt{L}(s_2); s_2{\downarrow}n]$.

Verifying $\preceq_{ci}$ directly requires checking infinitely many contexts. The following intrinsic preorder captures the same information using only the expression itself, without quantifying over contexts.

Definition 3.10 (Distribution-Cost Improvement)**.** Let s,t be stochastic programs. We write $s \preceq_{di} t$ if:

$$\text{(i) } s \sim_d t, \qquad \text{and (ii) for all } n \in \mathbb{N}\text{: } \mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(t) \,|\, t{\downarrow}n]$$

We write $s \succ_{di} t$ if $s \preceq_{di} t$ and $t \preceq_{di} s$.

Condition (i) ensures that replacing s by t does not change the distribution, and condition (ii) ensures that there is no expected increase in reduction length for every value.

Since the improvement preorders only compare semantically equivalent programs (condition (i) of Definitions 3.4 and 3.10), they restrict attention to programs with the same output distribution and hence the same convergence probability $\mathbb{P}(s{\downarrow})$. In particular, *AST status* (see Example 3.3) is preserved: if $s \preceq_{ci} t$ or $s \preceq_{di} t$, then $\mathbb{P}(s{\downarrow}) = \mathbb{P}(t{\downarrow})$, so s is AST iff t is. *PAST status*, however, may differ between improvement-related programs: PAST requires additionally $\mathbb{E}[\mathtt{L}(s)] < \infty$, which is precisely the cost component being compared. Two distribution-equivalent programs can have the same AST status while one is PAST and the other is not (one with finite $\mathbb{E}[\mathtt{L}(s)]$ and one with $\mathbb{E}[\mathtt{L}(s)] = \infty$). Thus the improvement preorders do not preserve PAST status in general.

We end this section by discussing other alternatives for the distribution-cost improvement.

1. Using $\mathbb{E}[\mathtt{L}(s);s{\downarrow}n]$ instead of $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}n]$: One might ask what happens if in Definition 3.10 one replaces the conditional expected evaluation length $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}n]$ by the expected evaluation length $\mathbb{E}[\mathtt{L}(s);s{\downarrow}n]$ directly: $s \lesssim_{el} t$ iff $s \sim_d t$ and $\mathbb{E}[\mathtt{L}(s);s{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(t);t{\downarrow}n]$ for all n. Since $\sim_d$ gives $\mathbb{P}(s{\downarrow}n) = \mathbb{P}(t{\downarrow}n)$ for all n, and $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}n] = \mathbb{E}[\mathtt{L}(s);s{\downarrow}n]/\mathbb{P}(s{\downarrow}n)$, dividing by the common positive factor $\mathbb{P}(s{\downarrow}n)$ shows that $\mathbb{E}[\mathtt{L}(s);s{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(t);t{\downarrow}n]$ and $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(t)\,|\,t{\downarrow}n]$ are equal (for n with $\mathbb{P}(s{\downarrow}n) > 0$). Hence $\lesssim_{el} = \preceq_{di}$.
2. Total expected cost improvement: A weaker variant replaces the per-value condition by an inequality on the *total* expected evaluation length: Let $s \lesssim_{tec} t$ iff $s \sim_d t$ and $\mathbb{E}[\mathtt{L}(s)] \geq \mathbb{E}[\mathtt{L}(t)]$. Since $\mathbb{E}[\mathtt{L}(s)] = \sum_n \mathbb{P}(s{\downarrow}n) \cdot \mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}n]$ (Proposition 3.7), this averages the per-value costs into a single number. It is strictly weaker than $\preceq_{di}$: $\preceq_{di}$ implies $\lesssim_{tec}$ (sum the per-value inequalities), but not vice versa. Let $\mathtt{pred}^k\, m$ denote k applications of $\mathtt{pred}$ to numeral m, and let $s = 0 \oplus \mathtt{pred}^8\, 9$ and $t = \mathtt{pred}^3\, 3 \oplus \mathtt{pred}^4\, 5$. Both produce 0 and 1 each with probability 0.5. For s: the path to 0 costs 1 step, and the path to 1 costs $1+8=9$ steps , giving $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}0] = 1$ and $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}1] = 9$. For t: the path to 0 costs $1+3=4$ steps and the path to 1 costs $1+4=5$ steps, giving $\mathbb{E}[\mathtt{L}(t)\,|\,t{\downarrow}0] = 4$ and $\mathbb{E}[\mathtt{L}(t)\,|\,t{\downarrow}1] = 5$. Then $\mathbb{E}[\mathtt{L}(s)] = 0.5 \cdot 1 + 0.5 \cdot 9 = 5 \geq 4.5 = \mathbb{E}[\mathtt{L}(t)]$, so $s \lesssim_{tec} t$. However, $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}0] = 1 < 4 = \mathbb{E}[\mathtt{L}(t)\,|\,t{\downarrow}0]$, so $s \not\preceq_{di} t$. Moreover, $s \not\preceq_{ci} t$: the context $C[\cdot] = \mathtt{let}\ x{=}[\cdot]\ \mathtt{in}\ \mathtt{if}\ x\ \mathtt{then}\ 0\ \mathtt{else}\ \mathtt{fix}(\lambda f.f)$ diverges unless $x = 0$; in the convergent case it adds one if-step (the intermediate *cp*-step is not counted), giving $\mathbb{E}[\mathtt{L}(C[s])] = 0.5 \cdot (1+1) = 1 < 2.5 = 0.5 \cdot (4+1) = \mathbb{E}[\mathtt{L}(C[t])]$. Hence $\lesssim_{tec} \not\subseteq \preceq_{ci}$.
3. Stochastic dominance on costs. Instead of comparing the conditional *mean* length $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}n]$, one may compare the whole conditional cost *distribution*. For $r \in \{s,t\}$, let us define $L_r(n,k) := \sum_{S:\, r{\downarrow}n,\, \mathtt{L}(r,S)\leq k} \mathsf{P}(S)$ as the probability that r produces n using at most k counted steps. Then define $s \lesssim_{sd} t$ iff $s \sim_d t$ and $L_s(n,k) \leq L_t(n,k)$ for all n,k, i.e. t produces n within k steps with at least as high a probability as s. Since dominance at every length bound k entails an ordering of the means, $\lesssim_{sd}$ implies $\preceq_{di}$. The converse fails because $\preceq_{di}$ constrains only the mean. A concrete witness is $s = \mathtt{pred}^2\, 2 \oplus \mathtt{pred}^4\, 4$ and $t = \mathtt{pred}^4\, 4$. Both converge to 0 with probability 1, hence $s \sim_d t$. The two paths of s cost $1+2=3$ and $1+4=5$ counted steps, so $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}0] = \frac{1}{2}(3+5) = 4$, while t deterministically costs 4 steps, so $\mathbb{E}[\mathtt{L}(t)\,|\,t{\downarrow}0] = 4$. Thus $s \asymp_{di} t$: the two programs are indistinguishable for $\preceq_{di}$. Their cost distributions differ nonetheless: t always finishes in exactly 4 steps, whereas s finishes in 3 steps with probability $\frac{1}{2}$ and in 5 steps with probability $\frac{1}{2}$. Concretely, $L_s(0,3) = \frac{1}{2} > 0 = L_t(0,3)$, so $s \not\lesssim_{sd} t$, while $L_t(0,4) = 1 > \frac{1}{2} = L_s(0,4)$, so $t \not\lesssim_{sd} s$. Hence two $\preceq_{di}$-equivalent programs can be incomparable under $\lesssim_{sd}$, witnessing the strict inclusion $\lesssim_{sd} \subsetneq \preceq_{di}$.

In summary, we have $\lesssim_{sd} \subsetneq \preceq_{di} = \lesssim_{el} \subsetneq \lesssim_{tec}$.

4 Main Theorem

Our main theorem states $\preceq_{ci} = \preceq_{di}$ for stochastic programs. We explain the proof strategy: Completeness ($\preceq_{ci} \subseteq \preceq_{di}$) is the easier direction. The semantic equivalence part uses Theorem 2.7, and for the cost-part, we construct for each n_0 a context R_{n_0} that isolates the value n_0. Then $\mathbb{E}[\mathtt{L}(s)\,|\,s{\downarrow}n_0] \geq \mathbb{E}[\mathtt{L}(t)\,|\,t{\downarrow}n_0]$ follows from the $\preceq_{ci}$-hypothesis. Soundness ($\preceq_{di} \subseteq \preceq_{ci}$) is the technical core. Contextual equivalence $s \sim_c t$ follows from $s \sim_d t$ via Theorem 2.7. Showing $\mathbb{E}[\mathtt{L}(C[s])] \geq \mathbb{E}[\mathtt{L}(C[t])]$ for all closing C proceeds in three steps: (1) In Proposition 4.4, we show that $s \preceq_{di} t$ implies $R[s] \preceq_{di} R[t]$ for every closed reduction context R. (2) A context lemma (Theorem 4.8 and Corollary 4.9) then lifts step (1) to arbitrary closing contexts. (3) The parts are combined to prove soundness.

Given an expression $R[s]$ where R is a reduction context and s is a closed expression, standard reduction $\xrightarrow{sr}$ does not impose an order between reductions inside the plugged expression s and reductions in the surrounding context R: in particular, *lll*-steps may interleave arbitrarily. The *sf*-strategy of [34] imposes a "s-first" discipline: reduce s exhaustively to a value n, then continue with $R[n]$.

Definition 4.1 (*sf*-Evaluation [34])**.** Let $R[\cdot_{nat}] : nat$ be a closed reduction context and $s_0 : nat$ a closed expression. An *sf-evaluation* of $R[s_0]$ first reduces s_0 exhaustively ($s_0 \xrightarrow{sr} \cdots \xrightarrow{sr} s_k$ with $s_k \not\xrightarrow{sr}$, phase 1), then continues with $R[s_k] = t_0 \xrightarrow{sr} \cdots \xrightarrow{sr} t_m$ until t_m is a WHNF (phase 2).

After phase 1, $s_k = \mathrm{LR}[n]$ for some $n \in \mathbb{N}$. Since n is a number, the bindings in LR are "garbage". Plugging $\mathrm{LR}[n]$ into R introduces only *lll*-steps, and thus, $\mathbb{P}(R[\mathrm{LR}[n]]{\downarrow}j) = \mathbb{P}(R[n]{\downarrow}j)$ and $\mathbb{E}[\mathtt{L}(R[\mathrm{LR}[n]]); R[\mathrm{LR}[n]]{\downarrow}j] = \mathbb{E}[\mathtt{L}(R[n]); R[n]{\downarrow}j]$. For readability, we write $R[n]$ instead of $R[\mathrm{LR}[n]]$.

Proposition 4.2 ([34, Prop. 4.14])**.** Let $s : nat$ be closed, $R[\cdot_{nat}] : nat$ be a closed reduction context. Every *sr*-evaluation of $R[s]$ has a corresponding *sf*-evaluation with the same final value, same probability, and the same sequence of non-*lll* reduction labels, and vice versa.

Corollary 4.3. $\mathbb{P}(R[s]{\downarrow}n)$ and $\mathbb{E}[\mathtt{L}(R[s]); R[s]{\downarrow}n]$ are the same under $\xrightarrow{sr}$ and $\xrightarrow{sf}$.

The first step of the soundness proof is to show that $\preceq_{di}$ is preserved when expressions are plugged into reduction contexts. The key idea is to use the *sf*-strategy to decompose the cost of $R[s]$ into a term cost (from evaluating s) and a context cost (from evaluating the surrounding R).

Proposition 4.4. Let $s_1 \preceq_{di} s_2$ and $R[\cdot_{nat}] : nat$ be a closed reduction context. Then $R[s_1] \preceq_{di} R[s_2]$.

Proof. Condition (i) ($R[s_1] \sim_d R[s_2]$) follows from $s_1 \sim_d s_2$ and Lemma 2.6. To prove (ii), it suffices to show $\mathbb{E}[\mathtt{L}(R[s_1]); R[s_1]{\downarrow}j] \geq \mathbb{E}[\mathtt{L}(R[s_2]); R[s_2]{\downarrow}j]$ (since the $\mathbb{P}(R[s_i]{\downarrow}j)$-values agree). By Corollary 4.3, we may compute under the *sf*-strategy and write $R[n]$ for $R[\mathrm{LR}[n]]$, giving (see Appendix A for details):

$$\mathbb{E}[\mathtt{L}(R[s_i]); R[s_i]{\downarrow}j] = \underbrace{\textstyle\sum_{n\geq 0} \mathbb{P}(s_i{\downarrow}n) \cdot \mathbb{E}[\mathtt{L}(R[n]); R[n]{\downarrow}j]}_{\text{phase-2 cost}} + \underbrace{\textstyle\sum_{n\geq 0} \mathbb{E}[\mathtt{L}(s_i); s_i{\downarrow}n] \cdot \mathbb{P}(R[n]{\downarrow}j)}_{\text{phase-1 cost}} \qquad (1)$$

Since $s_1 \preceq_{di} s_2$, $\mathbb{P}(s_1{\downarrow}n) = \mathbb{P}(s_2{\downarrow}n)$ (phase-2 cost are equal for s_1 and s_2) and $\mathbb{E}[\mathtt{L}(s_1); s_1{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(s_2); s_2{\downarrow}n]$ (phase-1 cost for $s_1 \geq$ phase-1 cost for s_2, since $\mathbb{P}(R[n]{\downarrow}j) \geq 0$). □

Let us recall that $\sim_d$-closure under closing reduction contexts lifts to closing multi-contexts:

Proposition 4.5 ([28, Prop. 4.6])**.** Let $C[\cdot_{1,\sigma}, \ldots, \cdot_{N,\sigma}] : nat$ be a multi-context and $s, t : \sigma$, such that $FV(C[s,\ldots,s]) = FV(C[t,\ldots,t]) = \emptyset$. If $R[s] \sim_d R[t]$ for all reduction contexts $R[\cdot_\sigma] : nat$, with $FV(R[s]) = FV(R[t]) = \emptyset$, then $C[s,\ldots,s] \leq_d C[t,\ldots,t]$.

For the cost part, we use the following bounded variants, which enable us to perform inductive proofs:

Definition 4.6 (Bounded Expected Convergence and Evaluation Length)**.** For stochastic program $s : nat$, $n \in \mathbb{N}$, and $k \geq 0$, let:

$$\mathbb{P}^{\leq k}(s{\downarrow}n) := \sum_{S : s{\downarrow}n, |S| \leq k} \mathsf{P}(S) \qquad\qquad \mathbb{E}^{\leq k}[\mathtt{L}(s); s{\downarrow}n] := \sum_{S : s{\downarrow}n, |S| \leq k} \mathtt{L}(s,S) \cdot \mathsf{P}(S),$$

By monotone convergence for non-negative series, we immediately have:

Lemma 4.7. The series $(\mathbb{P}^{\leq k}(s{\downarrow}n))_{k\geq 0}$ and $(\mathbb{E}^{\leq k}[\mathtt{L}(s); s{\downarrow}n])_{k\geq 0}$ are both non-decreasing with limits $\lim_{k\to\infty} \mathbb{P}^{\leq k}(s{\downarrow}n) = \mathbb{P}(s{\downarrow}n)$ and $\lim_{k\to\infty} \mathbb{E}^{\leq k}[\mathtt{L}(s); s{\downarrow}n] = \mathbb{E}[\mathtt{L}(s); s{\downarrow}n]$.

With these preparations, we can prove the core of the soundness argument:

Theorem 4.8 (Context Lemma for $\preceq_{di}$)**.** *Let s,t be stochastic programs with $s \preceq_{di} t$. Suppose that for all closed reduction contexts $R[\cdot_{nat}] : nat$: $R[s] \preceq_{di} R[t]$. Then $C[s,\ldots,s] \preceq_{di} C[t,\ldots,t]$ for all closed multi-contexts $C[\cdot_1,\ldots,\cdot_N] : nat$ (in particular $C[s] \preceq_{di} C[t]$ for all closing contexts $C[\cdot_{nat}] : nat$).*

Proof sketch. By Lemma 2.6 and applying Proposition 4.5 in both directions, we derive the equation $\mathbb{P}(C[s,\ldots,s]{\downarrow}n) = \mathbb{P}(C[t,\ldots,t]{\downarrow}n)$ for all C,n. It suffices to show the asymmetric strengthening $\mathbb{E}[\mathtt{L}(C[s,\ldots,s]);C[s,\ldots,s]{\downarrow}n] \geq \mathbb{E}^{\leq k}[\mathtt{L}(C[t,\ldots,t]);C[t,\ldots,t]{\downarrow}n]$ for all n,k (pass to the limit via Lemma 4.7). This is proved by lexicographic induction on (k,l,N), with prob-step bound k, deterministic steps to the next prob-reduction l, and the number N of holes, distinguishing whether C is a WHNF, a reduction context for some hole (chain the hypothesis with $N-1$ induction hypothesis), or neither (both sides take the same step). See Appendix B for details. □

Combining the context lemma with Proposition 4.4 yields the desired closure property:

Corollary 4.9. Let s,t be stochastic programs with $s \preceq_{di} t$. Then $C[s] \preceq_{di} C[t]$ for all closed $C[\cdot_{nat}] : nat$.

We are now able to prove our main theorem.

Main Theorem 4.10. For stochastic programs s,t: $s \preceq_{ci} t \iff s \preceq_{di} t$.

Proof. For completeness ($\preceq_{ci} \subseteq \preceq_{di}$), assume $s \preceq_{ci} t$. Then $s \sim_c t$ and $\mathbb{E}[\mathtt{L}(C[s])] \geq \mathbb{E}[\mathtt{L}(C[t])]$ for all closing contexts $C[\cdot_\sigma] : nat$. Theorem 2.7 implies $s \sim_d t$ and $\mathbb{P}(s{\downarrow}m) = \mathbb{P}(t{\downarrow}m)$ for all $m \in \mathbb{N}$. It suffices to show $\mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}m] \geq \mathbb{E}[\mathtt{L}(t) \,|\, t{\downarrow}m]$ for each $m \in \mathbb{N}$ with $\mathbb{P}(s{\downarrow}m) > 0$. To this end, fix m and let $\Omega = \mathtt{fix}(\lambda f.f)$. Define the family of expressions (parameterised by z):

$$E_0(z) \;:=\; \mathtt{if}\ z\ \mathtt{then}\ 0\ \mathtt{else}\ \Omega, \qquad E_{m+1}(z) \;:=\; \mathtt{if}\ z\ \mathtt{then}\ \Omega\ \mathtt{else}\ E_m(\mathtt{pred}\ z),$$

and set $R_m := \mathtt{let}\ x{=}[\cdot]\ \mathtt{in}\ E_m(x)$. Then, for each m, the closed expression $\mathtt{let}\ x{=}r\ \mathtt{in}\ E_m(x)$ has the following property: an evaluation of $\mathtt{let}\ x{=}r\ \mathtt{in}\ E_m(x)$ converges (to 0) if and only if the corresponding phase-1 evaluation of r yields the value m. When it does converge, it takes exactly k_m counted steps after the phase-1 evaluation of r, for some constant k_m depending only on m. This claim is proved by induction on m: if r evaluates to m, then $E_m(x)$ takes the appropriate if-branch at each level, propagates $\mathtt{pred}$ of x's value downward, and converges to 0 in a fixed number of steps; if r evaluates to any $m' \neq m$, then $E_m(x)$ eventually reaches Ω and diverges. Hence $\mathbb{E}[\mathtt{L}(R_m[s])] = \mathbb{E}[\mathtt{L}(s);s{\downarrow}m] + k_m \cdot \mathbb{P}(s{\downarrow}m)$ and likewise for t. Since $\mathbb{P}(s{\downarrow}m) = \mathbb{P}(t{\downarrow}m) > 0$ and $\mathbb{E}[\mathtt{L}(R_m[s])] \geq \mathbb{E}[\mathtt{L}(R_m[t])]$, we get $\mathbb{E}[\mathtt{L}(s);s{\downarrow}m] + k_m \cdot \mathbb{P}(s{\downarrow}m) \geq \mathbb{E}[\mathtt{L}(t);t{\downarrow}m] + k_m \cdot \mathbb{P}(t{\downarrow}m)$. Since $\mathbb{P}(s{\downarrow}m) = \mathbb{P}(t{\downarrow}m)$, the k_m-terms cancel, giving $\mathbb{E}[\mathtt{L}(s);s{\downarrow}m] \geq \mathbb{E}[\mathtt{L}(t);t{\downarrow}m]$, and dividing by $\mathbb{P}(s{\downarrow}m) > 0$ yields $\mathbb{E}[\mathtt{L}(s) \,|\, s{\downarrow}m] \geq \mathbb{E}[\mathtt{L}(t) \,|\, t{\downarrow}m]$.

For soundness ($\preceq_{di} \subseteq \preceq_{ci}$), let $s \preceq_{di} t$. Then $s \sim_c t$ (via $s \sim_d t$ and Theorem 2.7). By Corollary 4.9, $C[s] \preceq_{di} C[t]$ for every closing context C, so $\mathbb{P}(C[s]{\downarrow}n) = \mathbb{P}(C[t]{\downarrow}n)$ and $\mathbb{E}[\mathtt{L}(C[s]) \,|\, C[s]{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(C[t]) \,|\, C[t]{\downarrow}n]$ for all n. Multiplying (with the convention $0 \cdot \infty = 0$) gives $\mathbb{E}[\mathtt{L}(C[s]);C[s]{\downarrow}n] = \mathbb{P}(C[s]{\downarrow}n) \cdot \mathbb{E}[\mathtt{L}(C[s]) \,|\, C[s]{\downarrow}n] \geq \mathbb{P}(C[t]{\downarrow}n) \cdot \mathbb{E}[\mathtt{L}(C[t]) \,|\, C[t]{\downarrow}n] = \mathbb{E}[\mathtt{L}(C[t]);C[t]{\downarrow}n]$ for all n. By Proposition 3.7, $\mathbb{E}[\mathtt{L}(C[s])] = \sum_n \mathbb{E}[\mathtt{L}(C[s]);C[s]{\downarrow}n] \;\geq\; \sum_n \mathbb{E}[\mathtt{L}(C[t]);C[t]{\downarrow}n] = \mathbb{E}[\mathtt{L}(C[t])]$. □

5 Transformations and Examples

We consider exemplary program transformations. By Main Theorem 4.10 it suffices to verify each transformation as a $\preceq_{di}$-improvement; the contextual guarantee $\preceq_{ci}$ then follows automatically. We check the two conditions of Definition 3.10 directly: distribution equivalence and lower conditional expected cost per value (equivalently, $\mathbb{E}[\mathtt{L}(s);s{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(t);t{\downarrow}n]$ for all n by $\sim_d$).

As an example, consider a wheel of fortune with three results 1,2, and 3 all with the same probability. One encoding is the following program s_1 that uniformly chooses between 0,1,2, and 3 and recursively restarts the test if the result is 0: $s_1 := \texttt{fix}\ (\lambda f.\texttt{let}\ r = ((0 \oplus 1) \oplus (2 \oplus 3))\ \texttt{in}\ \texttt{if}\ r\ \texttt{then}\ f\ \texttt{else}\ r)$. A better encoding is the program $s_2 = 1 \overset{1/3}{\oplus} (2 \overset{1/2}{\oplus} 3)$. Both programs are semantically equal ($s_1 \sim_d s_2$), but $s_1 \preceq_{di} s_2$ (which is strict): For $i = 1,2$, $\mathbb{P}(s_i{\downarrow}n) = 1/3$ for $n \in \{1,2,3\}$ and $\mathbb{P}(s_i{\downarrow}n) = 0$ for $n \notin \{1,2,3\}$. Since there are two prob-steps for choosing a number from $\{0,1,2,3\}$, $\mathbb{E}[\mathtt{L}(s_1); s_1{\downarrow}n] \geq \sum_{k>0} 2k/4^k = 8/9$ for $n \in \{1,2,3\}$, $\mathbb{E}[\mathtt{L}(s_1); s_1{\downarrow}n] = 0$ for $n \notin \{1,2,3\}$, thus $\mathbb{E}[\mathtt{L}(s_1) \,|\, s_1{\downarrow}n] \geq 8/3$ for $n \in \{1,2,3\}$ and $\mathbb{E}[\mathtt{L}(s_1) \,|\, s_1{\downarrow}n] = \infty$ for $n \notin \{1,2,3\}$; $\mathbb{E}[\mathtt{L}(s_2); s_2{\downarrow}1] = 1/3, \mathbb{E}[\mathtt{L}(s_2); s_2{\downarrow}2] = 2/3, \mathbb{E}[\mathtt{L}(s_2); s_2{\downarrow}3] = 2/3, \mathbb{E}[\mathtt{L}(s_2); s_2{\downarrow}n] = 0$ for $n \notin \{1,2,3\}$ and $\mathbb{E}[\mathtt{L}(s_2) \,|\, s_2{\downarrow}1] = 1, \mathbb{E}[\mathtt{L}(s_2) \,|\, s_2{\downarrow}2] = 2, \mathbb{E}[\mathtt{L}(s_2) \,|\, s_2{\downarrow}3] = 2$, and $\mathbb{E}[\mathtt{L}(s_2) \,|\, s_2{\downarrow}n] = \infty$ for $n \notin \{1,2,3\}$.

Some algebraic laws of probabilistic choice [29, 34] hold as cost equivalences or strict improvements.

Proposition 5.1 (Algebraic laws for $\oplus$)**.** Let $r,s,t : \mathit{nat}$ be closed stochastic programs and $p,q \in (0,1)$.

(i) *Commutativity:* $s \overset{p}{\oplus} t \ \succ_{di}\ t \overset{1-p}{\oplus} s$.

(ii) *Idempotency:* $s \overset{p}{\oplus} s \ \preceq_{di}\ s$ (strictly whenever $\mathbb{P}(s{\downarrow}n) > 0$ for some n).

(iii) *Distributivity:* $(r \overset{p}{\oplus} s) \overset{q}{\oplus} (r \overset{p}{\oplus} t) \ \preceq_{di}\ r \overset{p}{\oplus} (s \overset{q}{\oplus} t)$ (strictly if $\mathbb{P}(r{\downarrow}n) > 0$ for some n).

Proof. All parts use Lemma 3.9 to compute the expected value convergence and expected evaluation length by unfolding the outermost prob-step.

(i) We have $\mathbb{P}(s \overset{p}{\oplus} t{\downarrow}n) = \mathbb{P}(t \overset{1-p}{\oplus} s{\downarrow}n) = p \cdot \mathbb{P}(s{\downarrow}n) + (1-p) \cdot \mathbb{P}(t{\downarrow}n) = u$, and $\mathbb{E}[\mathtt{L}(s \overset{p}{\oplus} t); s \overset{p}{\oplus} t{\downarrow}n] = \mathbb{E}[\mathtt{L}(t \overset{1-p}{\oplus} s); t \overset{1-p}{\oplus} s{\downarrow}n] = u + p \cdot \mathbb{E}[\mathtt{L}(s); s{\downarrow}n] + (1-p) \cdot \mathbb{E}[\mathtt{L}(t); t{\downarrow}n]$.

(ii) $\mathbb{P}(s \overset{p}{\oplus} s{\downarrow}n) = \mathbb{P}(s{\downarrow}n)$ by linearity, and $\mathbb{E}[\mathtt{L}(s \overset{p}{\oplus} s); s \overset{p}{\oplus} s{\downarrow}n] = \mathbb{P}(s{\downarrow}n) + \mathbb{E}[\mathtt{L}(s); s{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(s); s{\downarrow}n]$. Since, e.g., $\mathbb{E}[\mathtt{L}(1 \oplus 1) \,|\, 1 \oplus 1{\downarrow}1] = 1 > \mathbb{E}[\mathtt{L}(1) \,|\, 1{\downarrow}1] = 0$, the reverse is wrong.

(iii) Both sides assign probability p to r, $(1-p)q$ to s, and $(1-p)(1-q)$ to t, so $\sim_d$ holds. The left side takes two prob-steps to reach r or s, while the right side reaches r in one prob-step; unfolding with Lemma 3.9 shows $\mathbb{E}[\mathtt{L}(\mathrm{lhs}); \mathrm{lhs}{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(\mathrm{rhs}); \mathrm{rhs}{\downarrow}n]$. The reverse fails: e.g. $\mathbb{E}[\mathtt{L}((1 \oplus 2) \oplus (1 \oplus 3)) \,|\, (1 \oplus 2) \oplus (1 \oplus 3){\downarrow}1] = 2$ but $\mathbb{E}[\mathtt{L}(1 \oplus (2 \oplus 3)) \,|\, 1 \oplus (2 \oplus 3){\downarrow}1] = 1$. □

Associativity fails even for $\sim_d$ (it changes output probabilities [34]), so $\preceq_{di}$ fails in both directions.

We now focus on standard program transformations from lazy functional programming.

Proposition 5.2 (Garbage Collection and Let-floating)**.**

(i) *(GC)* Let $x \notin FV(t)$ and $\texttt{let}\ x{=}s\ \texttt{in}\ t$ a stochastic program. Then $\texttt{let}\ x{=}s\ \texttt{in}\ t \ \succ_{di}\ t$.

(ii) *(Let-float)* Let $x \notin FV(A^1)$ and $A^1[\texttt{let}\ x{=}s\ \texttt{in}\ t]$ a stochastic program. Then $A^1[\texttt{let}\ x{=}s\ \texttt{in}\ t] \ \succ_{di}\ \texttt{let}\ x{=}s\ \texttt{in}\ A^1[t]$.

Proof. (i) Since $x \notin FV(t)$, the binding $x{=}s$ is never demanded: the LR-context $\texttt{let}\ x{=}s\ \texttt{in}\ [\cdot]$ is carried passively throughout the evaluation of t. Every evaluation of $\texttt{let}\ x{=}s\ \texttt{in}\ t$ corresponds bijectively to an evaluation of t with the same prob-sequence, the same counted steps, and the same output value (the WHNFs differ only by the outer LR-hull, which does not affect *val*). Hence, expected value convergence and expected evaluation length coincide on both sides. (ii) The two sides are related by exactly one $(sr, lflata)$-step, which is a *lll*-step and thus not counted by L. Hence, expected value convergence and expected evaluation length are identical on both sides. □

Proposition 5.3 (Surface unique copying)**.** A context $C[\cdot_\sigma]$ is a *surface context* if the hole $[\cdot_\sigma]$ does not appear in the body of an abstraction. Let $C[\cdot_{nat}]$ be a *surface* closing context, and let t, $C[t]$ be stochastic programs with x a fresh variable. Then $\texttt{let } x{=}t \texttt{ in } C[x] \succeq_{di} C[t]$.

Proof. A let-bound variable x is evaluated at most once, and subsequent uses are then served by the *cp*-rule from the already-computed binding $\texttt{let } x{=}v \texttt{ in } \ldots$. Because x is fresh and C is a surface context, x's single occurrence is not guarded by any λ. Hence, x can only be reached by the reduction from a position that is directly demanded (i.e. inside a reduction context), and this happens at most once per evaluation path: once copied, $x = v$ is a value and no further evaluation of t is triggered. In particular, $C[t]$ and $\texttt{let } x{=}t \texttt{ in } C[x]$ evaluate t (x, respectively) at most once on every run, and produce the same output distribution: $\mathbb{P}(\texttt{let } x{=}t \texttt{ in } C[x]{\downarrow}n) = \mathbb{P}(C[t]{\downarrow}n)$ for all n. The `let`-version requires at most one additional (sr, cp)-step, however, *cp*-steps are not counted by `L` (Definition 3.1). Apart from this administrative step, the remaining reductions of $\texttt{let } x{=}v \texttt{ in } C[v]$ and $C[v]$ proceed step for step with the same rules and the same counted-step count. Hence $\mathbb{E}[\texttt{L}(\texttt{let } x{=}t \texttt{ in } C[x]);\texttt{let } x{=}t \texttt{ in } C[x]{\downarrow}n] = \mathbb{E}[\texttt{L}(C[t]);C[t]{\downarrow}n]$ for all n, and therefore $\texttt{let } x{=}t \texttt{ in } C[x] \succeq_{di} C[t]$. □

We discuss common subexpression elimination (CSE). For probabilistic s, sharing changes the output distribution: $r_1 = \texttt{if } (0 \oplus 1) \texttt{ then } (0 \oplus 1) \texttt{ else } 2$ has two *independent* coin flips giving $\mathbb{P}(r_1{\downarrow}0) = \mathbb{P}(r_1{\downarrow}1) = \frac{1}{4}, \mathbb{P}(r_1{\downarrow}2) = \frac{1}{2}$. The shared version $r_2 = \texttt{let } x{=}0 \oplus 1 \texttt{ in if } x \texttt{ then } x \texttt{ else } 2$ yields $\mathbb{P}(r_2{\downarrow}1) = 0$. So $r_1 \not\sim_d r_2$ and CSE is unsound for probabilistic subexpressions.

For deterministic s, both sides agree on $\sim_d$, and sharing avoids repeated evaluation:

Conjecture 5.4 (Deterministic CSE)**.** Let $C[\cdot_{1,\sigma}, \ldots, \cdot_{N,\sigma}] : nat$ be a *closing* multi-context with $N \geq 1$ holes, and let $s : \sigma$ be a closed, prob-free expression. Then $C[s, \ldots, s] \preceq_{di} \texttt{let } x{=}s \texttt{ in } C[x, \ldots, x]$.

The intuition is as follows: the shared version evaluates s exactly once on any run where x is demanded (costing c counted steps, if evaluation of s terminates), and propagates the value via *cp*-steps (not counted by `L`). In contrast, each of the (path-dependent) $k \leq N$ demanded copies of s in $C[s, \ldots, s]$ incurs the full c counted steps, giving a per-path difference of $(k-1) \cdot c \geq 0$ in favour of the shared version. Turning this argument into a proof requires a path-wise simulation for every terminating evaluation, similar to the proof in [33]. Adapting it to the probabilistic multi-context setting is left as future work.

6 Conclusion

We have developed an improvement theory for *ProbPCFR*^{need}, the probabilistic call-by-need lambda calculus. Distribution-cost improvement ($\preceq_{di}$) coincides with contextual cost improvement ($\preceq_{ci}$): for stochastic programs s, t, $s \preceq_{ci} t$ iff $s \preceq_{di} t$. The characterisation makes improvement verifiable without quantifying over all contexts. It suffices to check that t produces each output value with the same probability as s and, depending on each value, requires no more expected steps.

For future work, several directions are open. A *probabilistic tick algebra* [30] and a *bisimulation-based* proof technique for $\preceq_{di}$ would simplify verification of concrete improvements. On the language side, extending the characterisation to *higher types* and adding a `seq` operator for *strictness optimisations* are natural next steps. A complete proof of Conjecture 5.4 (deterministic CSE) is an open problem, requiring a path-wise simulation in the probabilistic multi-context setting, adapting [33]. Finally, *space improvements* as in [10] pose new challenges since space costs do not decompose additively.

Acknowledgements We thank the anonymous reviewers of WPTE 2026 for their careful reading and their very valuable comments.

References

[1] Martin Avanzini, Gilles Barthe & Ugo Dal Lago (2021): *On Continuation-Passing Transformations and Expected Cost Analysis*. *Proc. ACM Program. Lang.* 5(ICFP), pp. 87:1–87:30, doi:10.1145/3473592.

[2] Martin Avanzini, Ugo Dal Lago & Alexis Ghyselen (2019): *Type-Based Complexity Analysis of Probabilistic Functional Programs*. In: *Proc. LICS 2019*, IEEE, pp. 1–13, doi:10.1109/LICS.2019.8785725.

[3] Martin Avanzini, Georg Moser & Michael Schaper (2020): *A Modular Cost Analysis for Probabilistic Programs*. *Proc. ACM Program. Lang.* 4(OOPSLA), pp. 172:1–172:30, doi:10.1145/3428240.

[4] Pedro H. Azevedo de Amorim (2025): *Denotational Foundations for Expected Cost Analysis*. *Proc. ACM Program. Lang.* 9(OOPSLA1), pp. 280–306, doi:10.1145/3720424.

[5] Gilles Barthe, Joost-Pieter Katoen & Ana Silva, editors (2020): *Foundations of Probabilistic Programming*. Cambridge University Press, doi:10.1017/9781108770750.

[6] Aleš Bizjak & Lars Birkedal (2015): *Step-Indexed Logical Relations for Probability*. In: *Proc. FoSSaCS 2015*, *LNCS* 9034, Springer, pp. 279–294, doi:10.1007/978-3-662-46678-0_18.

[7] Martín Ceresa & Mauro Jaskelioff (2022): *Effectful improvement theory*. *Sci. Comput. Program.* 217, p. 102792, doi:10.1016/j.scico.2022.102792.

[8] Ugo Dal Lago, Davide Sangiorgi & Michele Alberti (2014): *On Coinductive Equivalences for Higher-Order Probabilistic Functional Programs*. In: *Proc. POPL 2014*, ACM, pp. 297–308, doi:10.1145/2535838.2535872.

[9] Thomas Ehrhard, Michele Pagani & Christine Tasson (2018): *Full Abstraction for Probabilistic PCF*. *J. ACM* 65(4), pp. 23:1–23:44, doi:10.1145/3164540.

[10] Jörgen Gustavsson & David Sands (2001): *Possibilities and Limitations of Call-by-Need Space Improvement*. In: *Proc. ICFP 2001*, ACM, pp. 265–276, doi:10.1145/507635.507667.

[11] Jennifer Hackett & Graham Hutton (2014): *Worker/wrapper/makes it/faster*. *SIGPLAN Not.* 49(9), pp. 95–107, doi:10.1145/2692915.2628142.

[12] Jennifer Hackett & Graham Hutton (2015): *Programs for Cheap!* In: *Proc. LICS 2015*, IEEE, pp. 115–126, doi:10.1109/LICS.2015.21.

[13] Jennifer Hackett & Graham Hutton (2018): *Parametric polymorphism and operational improvement*. *Proc. ACM Program. Lang.* 2(ICFP), pp. 68:1–68:24, doi:10.1145/3236763.

[14] Jennifer Hackett & Graham Hutton (2019): *Call-by-need is clairvoyant call-by-value*. *Proc. ACM Program. Lang.* 3(ICFP), pp. 114:1–114:23, doi:10.1145/3341718.

[15] Martin A. T. Handley, Niki Vazou & Graham Hutton (2020): *Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell*. *Proc. ACM Program. Lang.* 4(POPL), pp. 1–27, doi:10.1145/3371092.

[16] Benjamin Lucien Kaminski & Joost-Pieter Katoen (2015): *On the Hardness of Almost-Sure Termination*. In: *Proc. MFCS 2015*, *LNCS* 9234, Springer, pp. 307–318, doi:10.1007/978-3-662-48057-1_24.

[17] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja & Federico Olmedo (2018): *Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms*. *J. ACM* 65(5), pp. 30:1–30:68, doi:10.1145/3208102.

[18] Jan-Christoph Kassing, Leon Valentin Spitzer & Jürgen Giesl (2025): *Dependency Pairs for Expected Innermost Runtime Complexity and Strong Almost-Sure Termination of Probabilistic Term Rewriting*. In: *Proc. PPDP 2025*, ACM, pp. 10:1–10:14, doi:10.1145/3756907.3756917.

[19] Satoshi Kura & Hiroshi Unno (2024): *Automated Verification of Higher-Order Probabilistic Programs via a Dependent Refinement Type System*. *Proc. ACM Program. Lang.* 8(ICFP), pp. 973–1002, doi:10.1145/3674662.

[20] Rupak Majumdar & V. R. Sathiyanarayana (2024): *Positive Almost-Sure Termination: Complexity and Proof Rules*. *Proc. ACM Program. Lang.* 8(POPL), pp. 1089–1117, doi:10.1145/3632879.

[21] Fabian Meyer, Marcel Hark & Jürgen Giesl (2021): *Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes*. In: *Proc. TACAS 2021*, *LNCS* 12651, Springer, pp. 250–269, doi:10.1007/978-3-030-72016-2_14.

[22] Andrew Moran & David Sands (1999): *Improvement in a Lazy Context: An Operational Theory for Call-by-Need*. In: *Proc. POPL 1999*, ACM, pp. 43–56, doi:10.1145/292540.292547.

[23] Van Chan Ngo, Quentin Carbonneaux & Jan Hoffmann (2018): *Bounded Expectations: Resource Analysis for Probabilistic Programs*. In: *Proc. PLDI 2018*, ACM, pp. 496–512, doi:10.1145/3192366.3192394.

[24] Simon Peyton Jones, Will Partain & André Santos (1996): *Let-floating: moving bindings to give faster programs*. In: *Proc. ICFP 1996*, ACM, pp. 1–12, doi:10.1145/232627.232630.

[25] Simon Peyton Jones & André Santos (1998): *A transformation-based optimiser for Haskell*. *Sci. Comput. Program.* 32(1), pp. 3–47, doi:10.1016/S0167-6423(97)00029-4.

[26] David Sabel & Manfred Schmidt-Schauß (2008): *A call-by-need lambda calculus with locally bottom-avoiding choice: context lemma and correctness of transformations*. *Math. Structures Comput. Sci.* 18(3), pp. 501–553, doi:10.1017/S0960129508006774.

[27] David Sabel & Manfred Schmidt-Schauß (2016): *A Call-by-Need Lambda Calculus with Scoped Work Decorations*. In: *Proc. Software Engineering Workshops 2016*, *CEUR Workshop Proceedings* 1559, CEUR-WS.org, pp. 70–90. Available at `https://ceur-ws.org/Vol-1559/paper06.pdf`. ATPS 2016.

[28] David Sabel & Manfred Schmidt-Schauß (2025): *Probabilistic Lazy PCF with Real-Valued Choice*. Informal Proc. WPTE 2025. Available at `https://wpte2025.github.io/pre-proceedings.pdf#section*.9`.

[29] David Sabel, Manfred Schmidt-Schauß & Luca Maio (2022): *Contextual Equivalence in a Probabilistic Call-by-Need Lambda-Calculus*. In: *Proc. PPDP 2022*, ACM, pp. 4:1–4:15, doi:10.1145/3551357.3551374.

[30] David Sands (1996): *Total correctness by local improvement in the transformation of functional programs*. *ACM Trans. Program. Lang. Syst.* 18(2), pp. 175–234, doi:10.1145/227699.227716.

[31] Manfred Schmidt-Schauß & Nils Dallmeyer (2017): *Space Improvements and Equivalences in a Functional Core Language*. In: *Proc. WPTE 2017*, *EPTCS* 265, pp. 98–112, doi:10.4204/EPTCS.265.8.

[32] Manfred Schmidt-Schauß & David Sabel (2015): *Sharing-aware Improvements in a Call-by-need Functional Core Language*. In: *Proc. IFL 2015*, ACM, New York, NY, USA, pp. 6:1–6:12, doi:10.1145/2897336.2897343.

[33] Manfred Schmidt-Schauß & David Sabel (2017): *Improvements in a Call-by-Need Functional Core Language: Common Subexpression Elimination and Resource Preserving Translations*. *Sci. Comput. Program.* 147, pp. 3–26, doi:10.1016/j.scico.2017.04.003.

[34] Manfred Schmidt-Schauß & David Sabel (2023): *Program equivalence in a typed probabilistic call-by-need functional language*. *J. Log. Algebraic Methods Program.* 135, p. 100904, doi:10.1016/j.jlamp.2023.100904.

[35] Manfred Schmidt-Schauß, David Sabel & Nils Dallmeyer (2018): *Sequential and Parallel Improvements in a Concurrent Functional Programming Language*. In: *Proc. PPDP 2018*, ACM, pp. 20:1–20:13, doi:10.1145/3236950.3236952.

[36] Daniel Seidel & Janis Voigtländer (2011): *Improvements for Free*. In: *Proc. QAPL 2011*, *EPTCS* 57, pp. 89–103, doi:10.4204/EPTCS.57.7.

[37] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos & Andrew Cobb (2018): *Contextual Equivalence for a Probabilistic Language with Continuous Random Variables and Recursion*. *Proc. ACM Program. Lang.* 2(ICFP), pp. 87:1–87:30, doi:10.1145/3236782.

[38] Di Wang, David M. Kahn & Jan Hoffmann (2020): *Raising Expectations: Automating Expected Cost Analysis with Types*. *Proc. ACM Program. Lang.* 4(ICFP), pp. 110:1–110:31, doi:10.1145/3408992.

[39] Peixin Wang, Hongfei Fu, Amir Kafshdar Goharshady, Krishnendu Chatterjee, Xudong Qin & Wenjun Shi (2019): *Cost Analysis of Nondeterministic Probabilistic Programs*. In: *Proc. PLDI 2019*, ACM, pp. 204–220, doi:10.1145/3314221.3314581.

A Detailed Derivation of Equation (1)

We derive Equation (1) by unfolding $\mathbb{E}[\mathtt{L}(R[s_i]);R[s_i]{\downarrow}j]$ from the definition. Fix $i \in \{1,2\}$. Under the *sf*-strategy, every evaluation S of $R[s_i]$ to value j decomposes *uniquely* into a a phase-1 sequence S_1 reducing s_i to some WHNF $\mathrm{LR}[n]$, and a phase-2 sequence S_2 reducing $R[n]$ (short for $R[\mathrm{LR}[n]]$) to j. Since *lengths add*, i.e., $\mathtt{L}(R[s_i],S) = \mathtt{L}(s_i,S_1) + \mathtt{L}(R[n],S_2)$, and since *probabilities multiply*, i.e., $\mathsf{P}(S) = \mathsf{P}(S_1)\cdot\mathsf{P}(S_2)$, the computation can be rewritten as:

$$\mathbb{E}[\mathtt{L}(R[s_i]);R[s_i]{\downarrow}j] = \sum_{S:R[s_i]{\downarrow}j} \mathtt{L}(R[s_i],S)\cdot\mathsf{P}(S) = \sum_{n\geq 0}\sum_{S_1:s_i{\downarrow}n}\sum_{S_2:R[n]{\downarrow}j} \big(\mathtt{L}(s_i,S_1)+\mathtt{L}(R[n],S_2)\big)\cdot\mathsf{P}(S_1)\cdot\mathsf{P}(S_2).$$

Splitting the sum of lengths into two sums and factoring out the independent factors gives:

$$= \underbrace{\sum_{n\geq 0}\Big(\sum_{S_1:s_i{\downarrow}n}\mathtt{L}(s_i,S_1)\cdot\mathsf{P}(S_1)\Big)\Big(\sum_{S_2:R[n]{\downarrow}j}\mathsf{P}(S_2)\Big)}_{\text{term-cost sum }(A)} + \underbrace{\sum_{n\geq 0}\Big(\sum_{S_1:s_i{\downarrow}n}\mathsf{P}(S_1)\Big)\Big(\sum_{S_2:R[n]{\downarrow}j}\mathtt{L}(R[n],S_2)\cdot\mathsf{P}(S_2)\Big)}_{\text{context-cost sum }(B)}.$$

In sum (A), the inner sums over S_1 and S_2 depend on disjoint variables and can be evaluated separately. By definition, $\sum_{S_1:s_i{\downarrow}n}\mathtt{L}(s_i,S_1)\cdot\mathsf{P}(S_1) = \mathbb{E}[\mathtt{L}(s_i);s_i{\downarrow}n]$ and $\sum_{S_2:R[n]{\downarrow}j}\mathsf{P}(S_2) = \mathbb{P}(R[n]{\downarrow}j)$, giving $(A) = \sum_{n\geq 0}\mathbb{E}[\mathtt{L}(s_i);s_i{\downarrow}n]\cdot\mathbb{P}(R[n]{\downarrow}j)$. In sum (B) the S_1-sum yields $\mathbb{P}(s_i{\downarrow}n)$ and the S_2-sum yields $\mathbb{E}[\mathtt{L}(R[n]);R[n]{\downarrow}j]$, so $(B) = \sum_{n\geq 0}\mathbb{P}(s_i{\downarrow}n)\cdot\mathbb{E}[\mathtt{L}(R[n]);R[n]{\downarrow}j]$. Combining (A) and (B) yields (1).

B Proof of Theorem 4.8

As a preparation, we discover how a reduction step affects the bounded and unbounded cost quantities. A non-administrative deterministic step adds one to the length, an administrative *lll*-step leaves the length unchanged, and a prob-step splits the evaluation and decreases the prob-step bound k by one.

Lemma B.1. Let $s : \mathit{nat}$ be a closed expression.

(1) If $s \xrightarrow{sr,a} t$ with $a \notin \{\mathit{lll},\mathit{cp},\mathit{probl},\mathit{prob}\}$, then $\mathbb{P}^{\leq k}(s{\downarrow}n) = \mathbb{P}^{\leq k}(t{\downarrow}n)$ and $\mathbb{E}^{\leq k}[\mathtt{L}(s);s{\downarrow}n] = \mathbb{P}^{\leq k}(s{\downarrow}n) + \mathbb{E}^{\leq k}[\mathtt{L}(t);t{\downarrow}n]$.

(2) If $s \xrightarrow{sr,a} t$ with $a \in \{\mathit{lll},\mathit{cp}\}$, then $\mathbb{P}^{\leq k}(s{\downarrow}n) = \mathbb{P}^{\leq k}(t{\downarrow}n)$ and $\mathbb{E}^{\leq k}[\mathtt{L}(s);s{\downarrow}n] = \mathbb{E}^{\leq k}[\mathtt{L}(t);t{\downarrow}n]$.

(3) If $s \xrightarrow{sr,\mathit{probl}} s_1$ and $s \xrightarrow{sr,\mathit{probr}} s_2$ with probability p, then for $k = 0$, $\mathbb{P}^{\leq 0}(s{\downarrow}n) = 0$ and $\mathbb{E}^{\leq 0}[\mathtt{L}(s);s{\downarrow}n] = 0$, and for $k \geq 1$, $\mathbb{P}^{\leq k}(s{\downarrow}n) = p\cdot\mathbb{P}^{\leq k-1}(s_1{\downarrow}n) + (1-p)\cdot\mathbb{P}^{\leq k-1}(s_2{\downarrow}n)$ and $\mathbb{E}^{\leq k}[\mathtt{L}(s);s{\downarrow}n] = \mathbb{P}^{\leq k}(s{\downarrow}n) + p\cdot\mathbb{E}^{\leq k-1}[\mathtt{L}(s_1);s_1{\downarrow}n] + (1-p)\cdot\mathbb{E}^{\leq k-1}[\mathtt{L}(s_2);s_2{\downarrow}n]$.

Proof. Each case follows from the definition of $\mathbb{P}^{\leq k}(s{\downarrow}n)$ and $\mathbb{E}^{\leq k}[\mathtt{L}(s);s{\downarrow}n]$ by partitioning the evaluation sequences S according to their first step.

Case (1). Since the step $s \xrightarrow{sr,a} t$ is deterministic, every evaluation S of s to value n begins with this step. Since it is not a prob-reduction, the evaluation of t has the same prob-sequence S with probability $\mathsf{P}(S)$. We also have $\mathtt{L}(s,S) = 1 + \mathtt{L}(t,S)$ (since the step is counted by $\mathtt{L}$). Hence $\mathbb{P}^{\leq k}(s{\downarrow}n) = \sum_{S:s{\downarrow}n,|S|\leq k}\mathsf{P}(S) = \sum_{S:t{\downarrow}n,|S|\leq k}\mathsf{P}(S) = \mathbb{P}^{\leq k}(t{\downarrow}n)$ and $\mathbb{E}^{\leq k}[\mathtt{L}(s);s{\downarrow}n] = \sum_{S:s{\downarrow}n,|S|\leq k}\mathtt{L}(s,S)\cdot\mathsf{P}(S) = \sum_{S:t{\downarrow}n,|S|\leq k}(1+\mathtt{L}(t,S))\cdot\mathsf{P}(S) = \mathbb{P}^{\leq k}(t{\downarrow}n) + \mathbb{E}^{\leq k}[\mathtt{L}(t);t{\downarrow}n] = \mathbb{P}^{\leq k}(s{\downarrow}n) + \mathbb{E}^{\leq k}[\mathtt{L}(t);t{\downarrow}n]$

Case (2). As in case (1) but $\mathtt{L}(s,S) = \mathtt{L}(t,S)$, since *lll*- and *cp*-steps are not counted by $\mathtt{L}$. Hence $\mathbb{P}^{\leq k}(s{\downarrow}n) = \mathbb{P}^{\leq k}(t{\downarrow}n)$ and $\mathbb{E}^{\leq k}[\mathtt{L}(s);s{\downarrow}n] = \mathbb{E}^{\leq k}[\mathtt{L}(t);t{\downarrow}n]$.

Case (3). Every evaluation S of s to n with $|S| \leq k$ ($k \geq 1$) begins with the prob-step, leading to s_1 with weight p or to s_2 with weight $1-p$. If S starts with (x,q) where $(x,q) \in \{(\mathit{probl},p),(\mathit{probr},1-p)\}$ then

$S = (x,q), S'$ where $\mathsf{P}(S) = q \cdot \mathsf{P}(S')$, $|S'| = |S| - 1 \leq k-1$, and $\mathtt{L}(s,S) = 1 + \mathtt{L}(s_j, S')$ where s_j is s_1 if $x = \mathit{probl}$ and s_2 if $x = \mathit{probr}$. Hence

$$\mathbb{P}^{\leq k}(s{\downarrow}n) = \sum_{S: s{\downarrow}n, |S| \leq k} \mathsf{P}(S) = \sum_{S': s_1{\downarrow}n, |S'| \leq k-1} p \cdot \mathsf{P}(S') + \sum_{S': s_2{\downarrow}n, |S'| \leq k-1} (1-p) \cdot \mathsf{P}(S') = p \cdot \mathbb{P}^{\leq k-1}(s_1{\downarrow}n) + (1-p) \cdot \mathbb{P}^{\leq k-1}(s_2{\downarrow}n)$$

$$\begin{aligned}
\mathbb{E}^{\leq k}[\mathtt{L}(s); s{\downarrow}n] &= \sum_{S: s{\downarrow}n, |S| \leq k} \mathtt{L}(s,S) \cdot \mathsf{P}(S) = \sum_{S': s_1{\downarrow}n, |S'| \leq k-1} (1 + \mathtt{L}(s_1, S')) \cdot p \cdot \mathsf{P}(S') + \sum_{S': s_2{\downarrow}n, |S'| \leq k-1} (1 + \mathtt{L}(s_2, S')) \cdot (1-p) \cdot \mathsf{P}(S') \\
&= \sum_{S': s_1{\downarrow}n, |S'| \leq k-1} p \cdot \mathsf{P}(S') + \sum_{S': s_2{\downarrow}n, |S'| \leq k-1} (1-p) \cdot \mathsf{P}(S') + \sum_{S': s_1{\downarrow}n, |S'| \leq k-1} \mathtt{L}(s_1, S') \cdot p \cdot \mathsf{P}(S') + \sum_{S': s_2{\downarrow}n, |S'| \leq k-1} \mathtt{L}(s_2, S') \cdot (1-p) \cdot \mathsf{P}(S') \\
&= \mathbb{P}^{\leq k}(s{\downarrow}n) + p \cdot \mathbb{E}^{\leq k-1}[\mathtt{L}(s_1); s_1{\downarrow}n] + (1-p) \cdot \mathbb{E}^{\leq k-1}[\mathtt{L}(s_2); s_2{\downarrow}n].
\end{aligned}$$

For case $k = 0$, no evaluation with $|S| \leq 0$ reaches a WHNF. □

Corollary B.2 (Unbounded one-step unfolding)**.** Let $s : \mathit{nat}$ be a closed expression.

(1) If $s \xrightarrow{sr,a} t$ with $a \notin \{\mathit{lll}, \mathit{cp}, \mathit{probl}, \mathit{probr}\}$: $\mathbb{P}(s{\downarrow}n) = \mathbb{P}(t{\downarrow}n)$ and $\mathbb{E}[\mathtt{L}(s); s{\downarrow}n] = \mathbb{P}(s{\downarrow}n) + \mathbb{E}[\mathtt{L}(t); t{\downarrow}n]$.

(2) If $s \xrightarrow{sr,a} t$ with $a \in \{\mathit{lll}, \mathit{cp}\}$: $\mathbb{P}(s{\downarrow}n) = \mathbb{P}(t{\downarrow}n)$ and $\mathbb{E}[\mathtt{L}(s); s{\downarrow}n] = \mathbb{E}[\mathtt{L}(t); t{\downarrow}n]$.

(3) If the next step is a prob-step with probability p leading to s_1 (left) and s_2 (right): $\mathbb{P}(s{\downarrow}n) = p \cdot \mathbb{P}(s_1{\downarrow}n) + (1-p) \cdot \mathbb{P}(s_2{\downarrow}n)$ and $\mathbb{E}[\mathtt{L}(s); s{\downarrow}n] = \mathbb{P}(s{\downarrow}n) + p \cdot \mathbb{E}[\mathtt{L}(s_1); s_1{\downarrow}n] + (1-p) \cdot \mathbb{E}[\mathtt{L}(s_2); s_2{\downarrow}n]$.

With these preparations, we are able to give the full proof of Theorem 4.8.

Theorem 4.8 (Context Lemma for $\preceq_{di}$)**.** Let s, t be stochastic programs with $s \preceq_{di} t$. Suppose that for all closed reduction contexts $R[\cdot_{\mathit{nat}}] : \mathit{nat}$: $R[s] \preceq_{di} R[t]$. Then $C[s, \ldots, s] \preceq_{di} C[t, \ldots, t]$ for all closed multi-contexts $C[\cdot_1, \ldots, \cdot_N] : \mathit{nat}$ (in particular $C[s] \preceq_{di} C[t]$ for all closing contexts $C[\cdot_{\mathit{nat}}] : \mathit{nat}$).

Proof. Since $s \preceq_{di} t$ implies $s \sim_d t$, we have $\mathbb{P}(s{\downarrow}n) = \mathbb{P}(t{\downarrow}n)$ for all $n \in \mathbb{N}$. By Lemma 2.6, $R[s] \sim_d R[t]$ for all closed R, hence also $R[t] \sim_d R[s]$. Applying Proposition 4.5 with (s,t) yields $\mathbb{P}(C[s, \ldots, s]{\downarrow}n) \leq \mathbb{P}(C[t, \ldots, t]{\downarrow}n)$, and applying it with (t,s) yields the reverse inequality; hence for every multi-context C:

$$\mathbb{P}(C[s, \ldots, s]{\downarrow}n) \;=\; \mathbb{P}(C[t, \ldots, t]{\downarrow}n) \quad \text{for all } n. \tag{2}$$

Since the bounded version satisfies $\mathbb{P}^{\leq k}(C[t, \ldots, t]{\downarrow}n) \leq \mathbb{P}(C[t, \ldots, t]{\downarrow}n)$ for all k (by Lemma 4.7):

$$\mathbb{P}(C[s, \ldots, s]{\downarrow}n) \;=\; \mathbb{P}(C[t, \ldots, t]{\downarrow}n) \;\geq\; \mathbb{P}^{\leq k}(C[t, \ldots, t]{\downarrow}n) \quad \text{for all } n, k. \tag{3}$$

We show that $\mathbb{E}[\mathtt{L}(C[s, \ldots, s]); C[s, \ldots, s]{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(C[t, \ldots, t]); C[t, \ldots, t]{\downarrow}n]$ holds for all n. This is sufficient to prove $C[s, \ldots, s] \preceq_{di} C[t, \ldots, t]$, since the inequality on the conditional expected lengths follows by dividing by $\mathbb{P}(C[s, \ldots, s]{\downarrow}n) = \mathbb{P}(C[t, \ldots, t]{\downarrow}n)$. For proving the inequation on expected evaluation lengths, it suffices to show, for all $N \geq 0$ and all closed multi-contexts $C[\cdot_1, \ldots, \cdot_N] : \mathit{nat}$:

$$\mathbb{E}[\mathtt{L}(C[s, \ldots, s]); C[s, \ldots, s]{\downarrow}n] \;\geq\; \mathbb{E}^{\leq k}[\mathtt{L}(C[t, \ldots, t]); C[t, \ldots, t]{\downarrow}n] \quad \text{for all } n, k. \tag{4}$$

The series $(\mathbb{E}^{\leq k}[\mathtt{L}(C[t, \ldots, t]); C[t, \ldots, t]{\downarrow}n])_{k \geq 0}$ is monotonically non-decreasing and the limit of the series is $\mathbb{E}[\mathtt{L}(C[t, \ldots, t]); C[t, \ldots, t]{\downarrow}n]$. Thus, we can apply Lemma 4.7 (4) which shows the inequation $\mathbb{E}[\mathtt{L}(C[s, \ldots, s]); C[s, \ldots, s]{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(C[t, \ldots, t]); C[t, \ldots, t]{\downarrow}n]$. We prove Equation (4) by lexicographic induction on $(k, l, N) \in \mathbb{N}^3$, where k bounds the number of prob-reductions on the t-side, N is the number of holes in C, and $l \in \mathbb{N}$ is 0, if $C[t, \ldots, t]$ is a WHNF, or diverges deterministically (takes only deterministic *sr*-steps, never reaching a prob-redex or a WHNF), and otherwise, l is the number of deterministic *sr*-steps of $C[t, \ldots, t]$ until the next prob-reduction or until termination. Since l is always a natural number, the triple $(k, l, N) \in \mathbb{N}^3$ is well-founded under the lexicographic order.

Case $l = 0$. Three sub-cases arise: (a) $C[t,\ldots,t]$ is a WHNF. Then $\mathbb{E}^{\leq k}[\mathtt{L}(C[t,\ldots,t]);C[t,\ldots,t]{\downarrow}n] = 0$ for all n,k, so (4) holds since $\mathbb{E}[\mathtt{L}(C[s,\ldots,s]);C[s,\ldots,s]{\downarrow}n] \geq 0$. (b) $C[t,\ldots,t]$ diverges deterministically. Then $\mathbb{E}^{\leq k}[\mathtt{L}(C[t,\ldots,t]);C[t,\ldots,t]{\downarrow}n] = 0$ for all n,k, so (4) holds since $\mathbb{E}[\mathtt{L}(C[s,\ldots,s]);C[s,\ldots,s]{\downarrow}n] \geq 0$. (c) $k = 0$ and the next step of $C[t,\ldots,t]$ is a prob-step. Then $\mathbb{E}^{\leq 0}[\mathtt{L}(C[t,\ldots,t]);C[t,\ldots,t]{\downarrow}n] = 0$, so (4) holds trivially.

In the remaining cases $k+l \geq 1$ with $l \geq 1$ (for a deterministic next step) or $k \geq 1$ (for a prob next step), the triple strictly decreases.

Case $C[\cdot_1,\ldots,\cdot_N]$ *is a reduction context for some hole* j. Let $R^s_j := C[s,\ldots,s,\cdot_j,s,\ldots,s]$ and $C' := C[\cdot_1,\ldots,\cdot_{j-1},t,\cdot_{j+1},\ldots,\cdot_N]$. Then $R^s_j[s] = C[s,\ldots,s]$, $R^s_j[t] = C'[s,\ldots,s]$, and $C'[t,\ldots,t] = C[t,\ldots,t]$.

1. The hypothesis gives $R^s_j[s] \preceq_{di} R^s_j[t]$, hence $\mathbb{E}[\mathtt{L}(C[s,\ldots,s]);C[s,\ldots,s]{\downarrow}n] = \mathbb{E}[\mathtt{L}(R^s_j[s]);R^s_j[s]{\downarrow}n] \geq \mathbb{E}[\mathtt{L}(R^s_j[t]);R^s_j[t]{\downarrow}n] = \mathbb{E}[\mathtt{L}(C'[s,\ldots,s]);C'[s,\ldots,s]{\downarrow}n]$.
2. C' has $N-1$ holes. Since $(k,l,N-1) <_{lex} (k,l,N)$, the induction hypothesis is applicable: $\mathbb{E}[\mathtt{L}(C'[s,\ldots,s]);C'[s,\ldots,s]{\downarrow}n] \geq \mathbb{E}^{\leq k}[\mathtt{L}(C'[t,\ldots,t]);C'[t,\ldots,t]{\downarrow}n] = \mathbb{E}^{\leq k}[\mathtt{L}(C[t,\ldots,t]);C[t,\ldots,t]{\downarrow}n]$.

Combining Items 1 and 2 gives (4).

Case $C[\cdot_1,\ldots,\cdot_N]$ *is not a reduction context for any hole (and* $k+l \geq 1$*).*

1. If $l = 0$ and $k = 0$ and the next step is a prob-step: $\mathbb{E}^{\leq 0}[\mathtt{L}(C[t,\ldots,t]);C[t,\ldots,t]{\downarrow}n] = 0 \leq \mathbb{E}[\mathtt{L}(C[s,\ldots,s]);C[s,\ldots,s]{\downarrow}n]$.
2. Assume $k+l \geq 1$. Both sides take the same reduction step. We consider subcases:
   (a) The step is deterministic, so $C[t,\ldots,t] \xrightarrow{sr,a} C'[t,\ldots,t]$ and $C[s,\ldots,s] \xrightarrow{sr,a} C'[s,\ldots,s]$. Then l decreases, so $(k,l-1,N') <_{\text{lex}} (k,l,N)$.
   If $a \notin \{llet, lflata, cp\}$, then by Corollary B.2(1) and Lemma B.1(1):

$$\mathbb{E}[\mathtt{L}(C[s,\ldots,s]);C[s,\ldots,s]{\downarrow}n] = \mathbb{P}(C[s,\ldots,s]{\downarrow}n) + \mathbb{E}[\mathtt{L}(C'[s,\ldots,s]);C'[s,\ldots,s]{\downarrow}n],$$
$$\mathbb{E}^{\leq k}[\mathtt{L}(C[t,\ldots,t]);C[t,\ldots,t]{\downarrow}n] = \mathbb{P}^{\leq k}(C[t,\ldots,t]{\downarrow}n) + \mathbb{E}^{\leq k}[\mathtt{L}(C'[t,\ldots,t]);C'[t,\ldots,t]{\downarrow}n].$$

   We have $\mathbb{E}[\mathtt{L}(C'[s,\ldots,s]);C'[s,\ldots,s]{\downarrow}n] \geq \mathbb{E}^{\leq k}[\mathtt{L}(C'[t,\ldots,t]);C'[t,\ldots,t]{\downarrow}n]$ from the induction hypothesis. The inequation $\mathbb{P}(C[s,\ldots,s]{\downarrow}n) \geq \mathbb{P}^{\leq k}(C[t,\ldots,t]{\downarrow}n)$ holds by (3).
   If $a \in \{llet, lflata, cp\}$, then $\mathbb{E}[\mathtt{L}(C[s,\ldots,s]);C[s,\ldots,s]{\downarrow}n] = \mathbb{E}[\mathtt{L}(C'[s,\ldots,s]);C'[s,\ldots,s]{\downarrow}n]$ and $\mathbb{E}^{\leq k}[\mathtt{L}(C[t,\ldots,t]);C[t,\ldots,t]{\downarrow}n] = \mathbb{E}^{\leq k}[\mathtt{L}(C'[t,\ldots,t]);C'[t,\ldots,t]{\downarrow}n]$ by Corollary B.2(2) and Lemma B.1(2). Apply the induction hypothesis to C'.
3. The step is a prob-reduction ($k \geq 1$): Then $C[t,\ldots,t] \xrightarrow{sr,probl} C_1[t,\ldots,t]$ with probability p_1, and $C[t,\ldots,t] \xrightarrow{sr,probr} C_2[t,\ldots,t]$ with probability $p_2 = 1-p_1$, and likewise for $C[s,\ldots,s]$. By Corollary B.2(3) and Lemma B.1(3):

$$\mathbb{E}[\mathtt{L}(C[s,\ldots,s]);C[s,\ldots,s]{\downarrow}n] = \mathbb{P}(C[s,\ldots,s]{\downarrow}n) + \textstyle\sum_{i=1,2} p_i \mathbb{E}[\mathtt{L}(C_i[s,\ldots,s]);C_i[s,\ldots,s]{\downarrow}n]$$
$$\mathbb{E}^{\leq k}[\mathtt{L}(C[t,\ldots,t]);C[t,\ldots,t]{\downarrow}n] = \mathbb{P}^{\leq k}(C[t,\ldots,t]{\downarrow}n) + \textstyle\sum_{i=1,2} p_i \mathbb{E}^{\leq k-1}[\mathtt{L}(C_i[t,\ldots,t]);C_i[t,\ldots,t]{\downarrow}n]$$

   For $i = 1,2$, we have $\mathbb{E}[\mathtt{L}(C_i[s,\ldots,s]);C_i[s,\ldots,s]{\downarrow}n] \geq \mathbb{E}^{\leq k-1}[\mathtt{L}(C_i[t,\ldots,t]);C_i[t,\ldots,t]{\downarrow}n]$ by the induction hypothesis. Equation (3) gives $\mathbb{P}(C[s,\ldots,s]{\downarrow}n) \geq \mathbb{P}^{\leq k}(C[t,\ldots,t]{\downarrow}n)$. □

A Cyclic Proof System for Trace Formula Implication with Least and Greatest Fixpoints

Takumi Sato Koji Nakazawa
Graduate School of Informatics, Nagoya University, Nagoya, Japan
sato.takumi@sqlab.jp knak@i.nagoya-u.ac.jp

Trace logic is an expressive formalism for specifying the step-wise behaviors of recursive programs using state predicates, binary relations, sequential composition, and fixed points. By characterizing a program via its strongest trace formula, verification is reduced to checking the validity of trace formula implications. However, existing proof systems for trace logic are limited to finite traces and rely on explicit fixed-point induction rules, which pose challenges for automated proof search since invariants must be explicitly provided. In this work, we extend trace logic with greatest fixed points to express infinite, reactive behaviors. Furthermore, we propose a cyclic proof system for checking trace formula implications in this extended logic. By representing inductive and co-inductive reasoning as cyclic structures within the proof derivation, our system reduces the need for explicit invariants. Finally, we formally establish the soundness of our proposed system.

1 Introduction

1.1 Background

Trace Logic [9, 10] is an expressive specification logic designed to reason about the step-wise execution traces of programs. While traditional deductive verification often relies on Hoare logic [11] to specify pre- and post-conditions, trace logic captures the entire sequence of state transitions, making it suitable for verifying complex recursive behaviors. It is built upon first-order state predicates, binary relations modeling atomic state updates, the chop connective $\Phi^\frown\Psi$, and the least fixed-point operator μ. In this framework, a recursive program S is characterized by its *strongest trace formula* $stf(S)$, and verification is reduced to proving $stf(S) \vdash \Phi$, where Φ is the desired trace property. For finite executions, the semantics of $stf(S)$ coincides with the traces generated by the small-step semantics of the language Rec [9, 10]. Thus, trace logic supports a translate-and-verify approach: a program is first translated into $stf(S)$, after which the resulting trace-formula implication is proved logically.

However, the current formalization of trace logic faces two major limitations. First, it primarily considers terminating executions and finite traces using only μ, leaving infinite behaviors of reactive systems out of scope. Second, Heidler et al. [10] proposed a sequent calculus that relies on explicit fixed-point induction rules. Applying these rules during automated proof search is challenging because an appropriate invariant must be supplied when the rule is applied.

Cyclic Proof Systems [3] represent induction and coinduction by allowing certain leaves of a derivation to link back to identical internal nodes. Soundness is ensured by a decidable global trace condition requiring infinite progress along every infinite proof path [2, 7, 8]. Since proof cycles can encode invariants incrementally, cyclic systems are amenable to automation and have been applied, for example, to Separation Logic [1, 14].

Cohen and Rowe [5] also integrate induction and coinduction using proof cycles, but their logic is based on transitive closure and its dual, whereas our system handles general nested fixed points, state predicates,

atomic updates, and chop. Matching Logic supports inductive and coinductive reasoning over general patterns [4], while our logic is specialized to state traces and sequential composition. Circular reasoning also appears in symbolic execution and all-path reachability systems derived directly from operational semantics [12, 15]; in contrast, we reason about trace-formula implications after translation.

1.2 Contribution

In this work, we extend trace logic, propose a cyclic proof system for trace formula implication, and demonstrate its soundness.

Comparison with prior work on trace logic and cyclic proof systems is as follows:

- We extend the syntax and semantics of trace logic with the greatest fixed-point operator ν. While previous frameworks [10] focused on finite traces using μ, our extension also supports non-terminating infinite behaviors.
- We construct a cyclic proof system that uniformly handles both μ and ν. Instead of using the explicit induction rules of Heidler et al. [10], our system unfolds fixed points and tracks them along proof branches.
- We define an admissibility condition using the outermost variable unfolded infinitely often: a μ-variable on the left or a ν-variable on the right. We prove soundness by adapting the semantic signature method of Dax et al. [7] and Doumane [8] to chop and state semantics.

Our examples cover least fixed points, greatest fixed points, and their nesting. Completeness and implementation are left for future work.

The remainder of this paper is organized as follows. Section 2 describes the syntax and semantics of the extended trace logic. Section 3 defines the cyclic proof system and its admissibility condition. Section 4 presents proof examples, and Section 5 establishes soundness. Finally, Section 6 concludes the paper.

2 Syntax and Semantics of Trace Logic

In this section, we briefly review the syntax and semantics of trace formulas. The target programs of this logic are in the language Rec [9, 10], which consists of standard imperative `while`-programs extended with recursive procedure calls. Trace formulas specify the step-wise behavior of these programs as sets of state sequences.

2.1 Trace Logic for Infinite Traces

Let *Var* be a set of state variables, P a set of first-order state predicates, $\mathscr{R}$ a set of binary relations, and $\mathscr{X}$ a set of fixed-point variables. To express infinite reactive behaviors alongside finite terminating traces, we extend trace logic with the greatest fixed-point operator ν. The syntax of trace formulas is:

$$\Phi, \Psi ::= p \mid R \mid X \mid \Phi \vee \Psi \mid \Phi \wedge \Psi \mid \Phi \frown \Psi \mid \mu X.\Phi \mid \nu X.\Phi.$$

Here, $p \in P$, $R \in \mathscr{R}$, and $X \in \mathscr{X}$. Since trace formulas contain no negation, fixed-point variables occur only in monotone contexts. The operators $\mu X.\Phi$ and $\nu X.\Phi$ bind X in Φ.

Let *Val* be a set of values and S the set of program states assigning values in *Val* to variables in *Var*. Following [9, 10], we use the identity relation Id and substitution relation Sb^a_x:

$$Id = \{(s,s) \mid s \in S\}, \qquad Sb^a_x = \{(s, s[x \mapsto s(a)]) \mid s \in S\}.$$

Here, $s(a)$ is the value of a in s, and $s[x \mapsto v]$ differs from s only in assigning v to x. We also let $\mathit{true} \in P$ be satisfied by every state.

Let $State^*$ be the set of finite state sequences, including ε, $State^+ = State^* \setminus \{\varepsilon\}$, and $State^\omega$ the set of infinite state sequences. We write $State^{\leq\omega} = State^* \cup State^\omega$ and $State^\infty = State^+ \cup State^\omega$. For $\sigma_1 \in State^*$ and $\sigma_2 \in State^{\leq\omega}$, the concatenation $\sigma_1 \cdot \sigma_2$ is the ordinary sequence concatenation; in particular, it imposes no endpoint condition and is defined only when σ_1 is finite.

Since $State^\infty$ allows infinite sequences, the fixed-point approximations may not stabilize within ω steps. However, by standard fixed-point theory [13], for every monotone operator over a complete lattice, there exists a *closure ordinal* κ such that its sequence of approximations stabilizes at κ, i.e., the approximation reaches the corresponding least or greatest fixed point. We fix a common ordinal bound κ that is sufficiently large for all fixed-point operators considered below.

We employ transfinite ordinals beyond ω because we do not assume that the semantic operators induced by trace formulas are ω-continuous or ω-cocontinuous. Thus, although some formulas stabilize at ω or earlier, transfinite approximations are required for the general monotone fixed-point semantics.

Accordingly, we define an *approximation level* as a function $\zeta : \mathscr{X} \to \kappa + 1$, which assigns an ordinal approximation bound up to κ to each fixed-point variable.

The semantics of Φ under a valuation $\mathscr{V} : \mathscr{X} \to 2^{State^\infty}$ and approximation level ζ is defined as follows:

$$\begin{aligned}
[\![p]\!]^\zeta_\mathscr{V} &= \{s \cdot \sigma \in State^\infty \mid s \models p\}, & [\![\Phi_1 \wedge \Phi_2]\!]^\zeta_\mathscr{V} &= [\![\Phi_1]\!]^\zeta_\mathscr{V} \cap [\![\Phi_2]\!]^\zeta_\mathscr{V},\\
[\![R]\!]^\zeta_\mathscr{V} &= \{s \cdot s' \mid R(s,s')\}, & [\![\Phi_1 \vee \Phi_2]\!]^\zeta_\mathscr{V} &= [\![\Phi_1]\!]^\zeta_\mathscr{V} \cup [\![\Phi_2]\!]^\zeta_\mathscr{V},\\
[\![X]\!]^\zeta_\mathscr{V} &= \mathscr{V}(X),\\
[\![\Phi_1 \frown \Phi_2]\!]^\zeta_\mathscr{V} &= \{\sigma_1 \cdot s \cdot \sigma_2 \in State^\infty \mid \sigma_1 \cdot s \in [\![\Phi_1]\!]^\zeta_\mathscr{V},\ s \cdot \sigma_2 \in [\![\Phi_2]\!]^\zeta_\mathscr{V}\},\\
[\![\mu X.\Phi]\!]^\zeta_\mathscr{V} &= (F_{\Phi,X,\zeta})^{\zeta(X)}_\mu(\emptyset), & [\![\nu X.\Phi]\!]^\zeta_\mathscr{V} &= (F_{\Phi,X,\zeta})^{\zeta(X)}_\nu(State^\infty).
\end{aligned}$$

Here, $F_{\Phi,X,\zeta}(A) = [\![\Phi]\!]^\zeta_{\mathscr{V}[X\mapsto A]}$ for $A \subseteq State^\infty$. Its approximants are:

$$\begin{aligned}
F^0_\eta(A) &= A, & F^{\alpha+1}_\eta(A) &= F(F^\alpha_\eta(A)),\\
F^\lambda_\mu(\emptyset) &= \bigcup_{\alpha<\lambda} F^\alpha_\mu(\emptyset), & F^\lambda_\nu(State^\infty) &= \bigcap_{\alpha<\lambda} F^\alpha_\nu(State^\infty),
\end{aligned}$$

where $\eta \in \{\mu,\nu\}$ and λ is a limit ordinal. For the standard semantics, let $\zeta_\kappa(X) = \kappa$ for every $X \in \mathscr{X}$, and write $[\![\Phi]\!]_\mathscr{V}$ for $[\![\Phi]\!]^{\zeta_\kappa}_\mathscr{V}$.

2.2 Sequent Validity and Auxiliary Definitions

A sequent is an expression of the form $\Gamma \vdash \Delta$, where Γ and Δ are finite sets of trace formulas.

Definition 2.1 (Validity)**.** A sequent $\Gamma \vdash \Delta$ is *valid* if, for every valuation $\mathscr{V}$, the intersection of the semantics of all formulas in the antecedent is subsumed by the union of the semantics of all formulas in the succedent:

$$\bigcap_{\Phi\in\Gamma} [\![\Phi]\!]_\mathscr{V} \subseteq \bigcup_{\Psi\in\Delta} [\![\Psi]\!]_\mathscr{V}$$

If Γ or Δ is empty, $\bigcap\emptyset$ evaluates to the set of all traces $State^\infty$, and $\bigcup\emptyset$ evaluates to the empty set $\emptyset$.

To formalize our calculus, we introduce auxiliary notations. The predicate projection $P_\Gamma = \{p \mid p \in \Gamma \cap P\}$ extracts the current state condition (i.e., state predicates) from Γ. For example, if $\Gamma = \{x > 0, Sb^{x-1}_x\}$, then $P_\Gamma = \{x > 0\}$. The strongest postcondition $spc_{x:=a}(p)$ computes the state predicate holding immediately after executing Sb^a_x from a state satisfying p.

$$\text{(CUT)}\ \frac{\Gamma, p \vdash \Delta \qquad \Gamma, \bar{p} \vdash \Delta}{\Gamma \vdash \Delta} \qquad \text{(REL)}\ \frac{}{\Gamma, R \vdash R', \Delta}(\{(s,s') \in R \mid s \models P_\Gamma\} \subseteq R') \qquad \text{(PRED)}\ \frac{P_\Gamma \vdash q \qquad \Gamma, q \vdash \Delta}{\Gamma \vdash \Delta}$$

$$\text{(CH-ID)}\ \frac{P_\Gamma, Id \vdash \Psi_1 \qquad \cdots \qquad P_\Gamma, Id \vdash \Psi_n \qquad P_\Gamma, \Phi \vdash \Psi'_1, \ldots, \Psi'_n}{\Gamma, Id^\frown\Phi \vdash \Psi_1{}^\frown\Psi'_1, \ldots, \Psi_n{}^\frown\Psi'_n, \Delta}$$

$$\text{(CH-UPD)}\ \frac{P_\Gamma, Sb^a_x \vdash \Psi_1 \qquad \cdots \qquad P_\Gamma, Sb^a_x \vdash \Psi_n \qquad spc_{x:=a}(P_\Gamma), \Phi \vdash \Psi'_1, \ldots, \Psi'_n}{\Gamma, Sb^a_x{}^\frown\Phi \vdash \Psi_1{}^\frown\Psi'_1, \ldots, \Psi_n{}^\frown\Psi'_n, \Delta} \qquad \text{(END-ID)}\ \frac{P_\Gamma, Id \vdash \Psi_1 \qquad P_\Gamma \vdash \Psi_2}{\Gamma, Id \vdash \Psi_1{}^\frown\Psi_2}$$

$$\text{(END-UPD)}\ \frac{P_\Gamma, Sb^a_x \vdash \Psi_1 \qquad spc_{x:=a}(P_\Gamma) \vdash \Psi_2}{\Gamma, Sb^a_x \vdash \Psi_1{}^\frown\Psi_2} \qquad \text{(CH-PREDL)}\ \frac{\Gamma, p, p^\frown\Phi \vdash \Delta}{\Gamma, p^\frown\Phi \vdash \Delta} \qquad \text{(CH-PREDR)}\ \frac{\Gamma, q \vdash true^\frown\Psi, \Delta}{\Gamma, q \vdash q^\frown\Psi, \Delta}$$

$$\text{(CH-}\vee\text{L)}\ \frac{\Gamma, \Phi_1{}^\frown\Phi_3 \vdash \Delta \qquad \Gamma, \Phi_2{}^\frown\Phi_3 \vdash \Delta}{\Gamma, (\Phi_1 \vee \Phi_2)^\frown\Phi_3 \vdash \Delta} \qquad \text{(CH-}\wedge\text{L)}\ \frac{\Gamma, \Phi_1{}^\frown\Phi_3, \Phi_2{}^\frown\Phi_3 \vdash \Delta}{\Gamma, (\Phi_1 \wedge \Phi_2)^\frown\Phi_3 \vdash \Delta} \qquad \text{(CH-}\vee\text{R)}\ \frac{\Gamma \vdash \Psi_1{}^\frown\Psi_3, \Psi_2{}^\frown\Psi_3, \Delta}{\Gamma \vdash (\Psi_1 \vee \Psi_2)^\frown\Psi_3, \Delta}$$

$$\text{(UNFL)}\ \frac{\Gamma, \Phi[\eta X.\Phi/X] \vdash \Delta}{\Gamma, \eta X.\Phi \vdash \Delta} \qquad \text{(UNFR)}\ \frac{\Gamma \vdash \Psi[\eta X.\Psi/X], \Delta}{\Gamma \vdash \eta X.\Psi, \Delta} \qquad \text{(CH-UNFL)}\ \frac{\Gamma, (\Phi[\eta X.\Phi/X])^\frown\Phi' \vdash \Delta}{\Gamma, (\eta X.\Phi)^\frown\Phi' \vdash \Delta}$$

$$\text{(CH-UNFR)}\ \frac{\Gamma \vdash (\Psi[\eta X.\Psi/X])^\frown\Psi', \Delta}{\Gamma \vdash (\eta X.\Psi)^\frown\Psi', \Delta} \qquad \text{(ARB1)}\ \frac{\Gamma \vdash \Psi, \Delta}{\Gamma \vdash true^\frown\Psi, \Delta} \qquad \text{(ARB2)}\ \frac{\Gamma, \Phi_1{}^\frown\Phi_2 \vdash \Phi_1{}^\frown true^\frown\Psi, \Delta}{\Gamma, \Phi_1{}^\frown\Phi_2 \vdash true^\frown\Psi, \Delta}$$

Figure 1: Inference rules of the cyclic proof system (where $\eta \in \{\mu, \nu\}$).

3 Cyclic Proof System

In the standard sequent calculus for trace formula implication proposed by Heidler et al. [10], reasoning about least fixed points relies on explicit fixed-point induction. This is formalized by rules such as (FPI-ALT):

$$\text{(FPI-ALT)}\ \frac{P_\Gamma \vdash I \qquad \xi, (X|_I, \Psi) \diamond I, \Phi \vdash \Psi}{\Gamma, \mu X.\Phi \vdash \Psi, \Delta}.$$

Here, I is an explicit state formula acting as an inductive invariant. The context ξ stores recursion contracts, and $\diamond$ acts as a separator between the context and the sequent. The notation $(X|_I, \Psi)$ adds a contract to the context ξ, meaning "evaluating X from a state satisfying I guarantees the succedent Ψ".

While this rule is sound, applying it during automated proof search is difficult. It requires explicitly providing the correct invariant I and the corresponding contract when the rule is applied, which acts as a significant hurdle for automation.

To eliminate the need for explicit invariants and make the calculus more amenable to automated proof search, we propose a *cyclic proof system*. In our system, explicit induction rules like (FPI-ALT) are entirely removed. Instead, fixed points are simply unfolded using the inference rules ((UNFL), (CH-UNFL), (UNFR), and (CH-UNFR)) presented in Figure 1. Before detailing the formal definitions, we briefly explain the intuition behind the core inference rules. The (CH-UPD) and (CH-ID) rules structurally decompose trace

concatenation ($\frown$) by advancing the state evaluation step-by-step. The unfolding rules ((UNFL), (UNFR)) and their chopped variants unroll the fixed-point bodies. Logical rules like (CUT) and (PRED) distribute the state semantics across branches. Specifically, we completely remove explicit fixed-point induction rules and adopt the remaining core inference rules (structural, logical, and trace-concatenation rules) from the trace logic sequent calculus introduced by Heidler et al. [10]. We naturally extend them to accommodate the greatest fixed-point operator ν, using $\eta \in \{\mu, \nu\}$ as a generic binder to denote both fixed-point operators uniformly. Furthermore, standard structural and logical rules of first-order sequent calculus (e.g., standard $\wedge L$ and $\vee R$) are assumed to be included in the system. The inductive and co-inductive reasoning is then captured globally by the cyclic structure of the proof graph.

3.1 Definition of Cyclic Proof System

To establish a formal cyclic proof system, we first define the graph structure of proofs and the conditions under which an infinite derivation is considered logically valid.

Definition 3.1 (Cyclic Pre-Proof)**.** A *cyclic pre-proof* is defined as a pair $(\mathscr{D}, F)$, where $\mathscr{D}$ is a finite derivation tree constructed from the inference rules in Figure 1, and F is a back-link function defined as follows. Let $\mathsf{Bud}(\mathscr{D})$ denote the set of non-axiom leaves (open premises) in $\mathscr{D}$. For every bud $B \in \mathsf{Bud}(\mathscr{D})$, the function F assigns it to an internal node $C = F(B)$ in $\mathscr{D}$ such that C is labeled with the exact same sequent as B. This mapping induces a finite directed graph with cycles.

However, not all pre-proofs are valid. To ensure validity, we trace the history of formulas along infinite paths. Note that in the context of cyclic proofs, we conventionally use the term *infinite path* to denote an infinite sequence of nodes (i.e., an infinite walk in graph theory) following the edges of this directed graph.

Definition 3.2 (Thread)**.** Let $\pi = (S_i)_{i\geq 0}$ be an infinite walk in a cyclic pre-proof $(\mathscr{D}, F)$, where each S_i is a sequent $\Gamma_i \vdash \Delta_i$. A sequence of trace formulas $\tau = (\Phi_i)_{i\geq 0}$ is called a *thread* along π if it satisfies the following conditions:

1. For every i, Φ_i is a direct ancestor of Φ_{i+1} with respect to the inference rule applied at S_i.
2. The sequence consistently tracks formulas on the same side of the turnstile, meaning either entirely within the antecedents ($\Phi_i \in \Gamma_i$ for all i) or entirely within the succedents ($\Phi_i \in \Delta_i$ for all i).
3. Specifically, an *unfolding point* for a fixed-point variable X is designated at S_i if Φ_{i+1} is the unfolded body of Φ_i resulting from the application of one of the following fixed-point unfolding rules at S_i:
   - For μ-variables: (UNFL), (CH-UNFL).
   - For ν-variables: (UNFR), (CH-UNFR).

A thread is called a *left-thread* if it tracks formulas in the antecedents, and a *right-thread* if it tracks formulas in the succedents.

Example 3.3. To clarify the notion of a direct ancestor, consider the application of the (CH-$\wedge$ L) rule.

$$\frac{\Gamma, \Phi_1 \frown \Phi_3, \Phi_2 \frown \Phi_3 \vdash \Delta}{\Gamma, (\Phi_1 \wedge \Phi_2) \frown \Phi_3 \vdash \Delta}$$

If a thread τ tracks the principal formula $(\Phi_1 \wedge \Phi_2) \frown \Phi_3$ in the conclusion (the lower sequent), its direct ancestor in the premise (the upper sequent) is either $\Phi_1 \frown \Phi_3$ or $\Phi_2 \frown \Phi_3$. The thread must choose exactly one of these two formulas to continue tracking. Similarly, for the (UNFL) rule, the direct ancestor of $\mu X.\Phi$ is its unfolded body $\Phi[\mu X.\Phi/X]$. For context formulas (those in Γ or Δ that are not modified by the rule), the direct ancestor is simply the identical formula in the premise.

Definition 3.4 (Dependency Order)**.** Without loss of generality, we assume that all bound variables in the formulas of the root sequent are mutually distinct. During fixed-point unfoldings, we strictly assume no α-renaming is performed. This convention ensures that structurally nested copies of bound variables identically refer to the same signature component. Let Φ be a trace formula. We define the dependency order $<_\Phi$ on the fixed-point variables bound in Φ. We say $X <_\Phi Y$ if the binding definition of Y (e.g., $\eta Y.\Psi$) is a proper subformula of the binding definition of X (e.g., $\eta X.\Psi'$). Essentially, $X <_\Phi Y$ means that X binds strictly outside Y in the original syntax tree.

Definition 3.5 (Admissible Thread)**.** Let $\tau = (\Phi_i)_{i\geq 0}$ be a thread along an infinite path π, originating from a formula Φ in the root sequent. Let $\mathrm{Inf}(\tau)$ be the set of fixed-point variables that have infinitely many unfolding points in τ. If $\mathrm{Inf}(\tau) \neq \emptyset$, then because variables unfolded infinitely often along a single thread must be structurally nested, $\mathrm{Inf}(\tau)$ is totally ordered with respect to $<_\Phi$. Thus, the minimum element $X_{out} = \min_{<_\Phi} \mathrm{Inf}(\tau)$ uniquely exists and is called the *outermost infinitely unfolded variable*. The thread τ is *admissible* if $\mathrm{Inf}(\tau) \neq \emptyset$ and it satisfies one of the following conditions:

- τ is a left-thread and X_{out} is a μ-variable.
- τ is a right-thread and X_{out} is a ν-variable.

This definition is based on the standard admissibility condition for cyclic proofs in fixed-point logics, as established in the works of Dax et al. [7] and Doumane [8].

Definition 3.6 (Cyclic Proof)**.** A cyclic pre-proof $(\mathscr{D}, F)$ is a cyclic proof if, for every infinite path π in $\mathscr{D}$, there exists at least one admissible thread τ along π.

4 Examples

In this section, we demonstrate our cyclic proof system using two examples: a reachability proof using the least fixed-point operator (μ) and an infinite safety verification using the greatest fixed-point operator (ν). Finally, we present a combined example featuring genuine μ/ν alternation.

4.1 Finite Trace Example: Reachability

Consider a terminating program S_{down} and its strongest trace formula (stf) [10], which exactly characterizes all possible execution traces of the program. Let $\Phi_{down} \equiv stf(S_{down})$ be defined as follows:

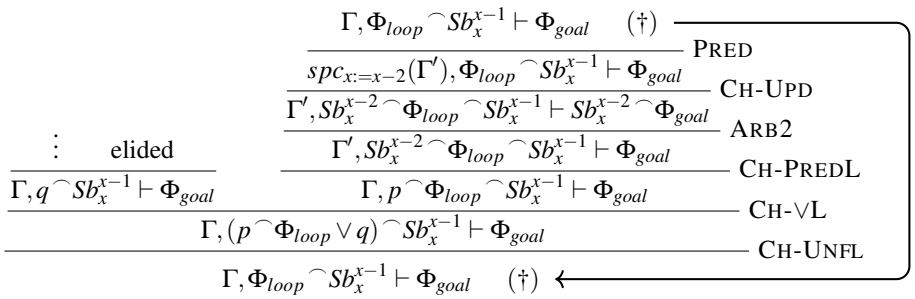
$$S_{down} : \texttt{while (x > 0) \{ x := x - 2; \}; x := x - 1;}$$
$$\Phi_{down} = (\mu X.(p \frown X \vee q)) \frown Sb_x^{x-1},$$

where $p \equiv x > 0 \frown Sb_x^{x-2}$ and $q \equiv x \leq 0 \frown Id$.

We verify that an initially non-negative even x eventually reaches $x = 0$ followed by $x = -1$. We define the specification $\Phi_{goal} = true \frown (x = 0 \frown x = -1)$ and prove the following entailment:

$$even(x), x \geq 0, \Phi_{down} \vdash \Phi_{goal}$$

The cyclic proof tree is shown in Figure 2. State conditions are automatically propagated via the strongest postcondition (*spc*), forming a cycle without explicit invariants. This is a proof because the left-thread tracking Φ_{loop} unfolds the μ-variable infinitely often via (CH-UNFL), yielding an admissible thread. For brevity, several intermediate logical steps and side premises (e.g., the state evaluation premises in CH-PREDL and CH-UPD) are elided in the displayed proof tree. Note that some routine side premises for state evaluations in CH-PREDL and CH-UPD are elided for brevity.



$$\Gamma \equiv even(x), x \geq 0 \quad \Gamma' \equiv even(x), x > 0 \quad \Phi_{loop} \equiv \mu X.(p \frown X \vee q)$$
$$p \equiv x > 0 \frown Sb_x^{x-2} \quad q \equiv x \leq 0 \frown Id \quad \Phi_{goal} \equiv true \frown (x = 0 \frown x = -1)$$

Figure 2: Cyclic proof tree for S_{down} reachability.

Furthermore, the elided q-branch in the proof tree represents the termination case of the loop. When evaluating $q \equiv x \leq 0 \frown Id$ under the context $\Gamma \equiv even(x), x \geq 0$, the state condition is constrained to exactly $x = 0$. Proceeding through the identity relation Id and the subsequent final assignment Sb_x^{x-1}, the execution naturally yields a trace that transitions from $x = 0$ to $x = -1$. This perfectly matches the right-hand side specification $\Phi_{goal} \equiv true \frown (x = 0 \frown x = -1)$, allowing this finite branch to be cleanly closed by the standard structural and relational rules without requiring any further unfoldings or back-links.

4.2 Infinite Trace Example: Safety Specification

Consider a non-terminating reactive program S_{alt} and its trace formula Φ_{alt}:

$$S_{alt} : \texttt{while (true) \{ x := 1; x := 0; \}}$$
$$\Phi_{alt} = \nu X.(Sb_x^1 \frown Sb_x^0 \frown X)$$

We want to verify a safety property ensuring that the value of x remains strictly less than 2. Specifically, we structurally express that the sequence of assignments $x := 1$ and $x := 0$ repeatedly occurs, and immediately after these assignments, the condition $x < 2$ holds continuously. The safety specification Φ_{safe} is defined as:

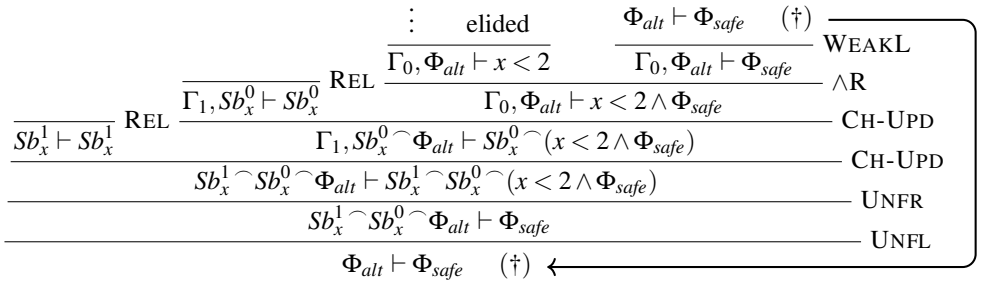
$$\Phi_{safe} = \nu Y.(Sb_x^1 \frown Sb_x^0 \frown (x < 2 \wedge Y))$$

The verification objective is to prove the entailment $\Phi_{alt} \vdash \Phi_{safe}$.

The cyclic proof tree is shown in Figure 3. Both the implementation and the specification are unfolded, and the assignment operations are structurally matched using the (CH-UPD) rule. As the proof progresses, the state condition is tracked via the strongest postcondition (spc): after executing $x := 1$ and $x := 0$, the updated context becomes $\Gamma_0 \equiv (x = 0)$. The standard right-conjunction rule ($\wedge$R) then splits the proof. The left branch verifies the state predicate $x < 2$ (which trivially holds since $x = 0$), and the right branch cycles back to the original goal, utilizing the structural weakening rule (WEAKL) to discard the accumulated state condition Γ_0 before the back-link is formed. Soundness is guaranteed because the right-thread tracking Φ_{safe} passes through the (UNFR) rule infinitely often, yielding an admissible thread.

4.3 Nested Alternation: Finite Reachability and Infinite Safety

To demonstrate the simultaneous treatment of μ-reachability and ν-safety within a single trace-logic-aware calculus, we consider a program S_{nest} featuring genuine fixed-point alternation. The program infinitely re-



$$\Phi_{alt} \equiv \nu X.(Sb_x^1 \frown Sb_x^0 \frown X) \quad \Phi_{safe} \equiv \nu Y.(Sb_x^1 \frown Sb_x^0 \frown (x < 2 \wedge Y))$$
$$\Gamma_1 \equiv (x = 1) \quad (\text{from } spc_{x:=1}(true)) \quad \Gamma_0 \equiv (x = 0) \quad (\text{from } spc_{x:=0}(\Gamma_1))$$

Figure 3: Cyclic proof tree for the S_{alt} safety specification.

peats a finite countdown phase followed by a reset phase:

$$S_{nest} : \texttt{while (true) \{ while (x > 0) \{ x := x - 1; \}; x := N; \}}$$
$$\Phi_{prog} = \nu Y.((\mu X.(p \frown X \vee q)) \frown Sb_x^N \frown Y)$$

where $p \equiv x > 0 \frown Sb_x^{x-1}$, $q \equiv x \leq 0 \frown Id$, and $N > 0$ is a strictly positive constant.

We verify that starting with $x \geq 0$, the program always completes the countdown (reaching $x = 0$) before resetting, and it repeats this alternating cycle infinitely. The specification is defined as:

$$\Phi_{spec} \equiv \nu V.((\mu U.(p \frown U \vee q \frown x = 0)) \frown Sb_x^N \frown V)$$

The objective is to prove the entailment: $x \geq 0, \Phi_{prog} \vdash \Phi_{spec}$.

The cyclic proof tree is shown in Figure 4. The proof yields a single graph with two distinct cycles: the inner μ-cycle ($\dagger_1$) verifying the finite countdown, and the outer ν-cycle ($\dagger_2$) verifying the infinite repetition.

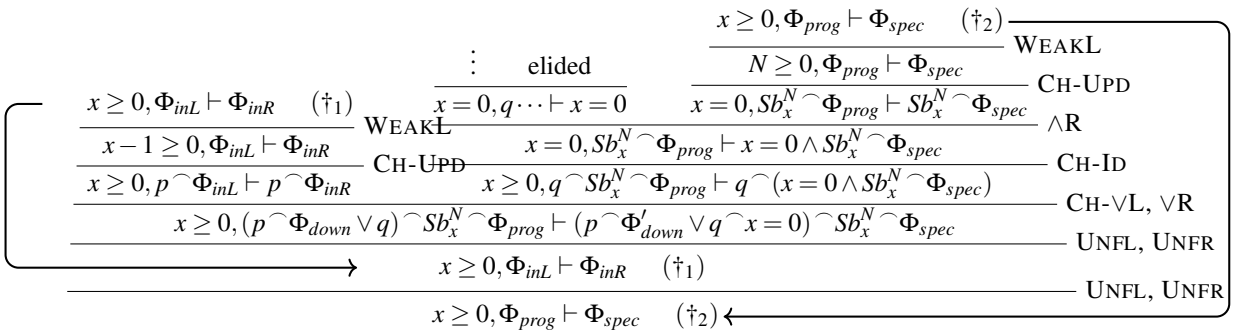
1. **The inner cycle** ($\dagger_1$): Evaluates the countdown loop. The left-thread tracking the μ-variable X is the only variable unfolded infinitely often in this cycle. Thus, it forms an admissible μ-thread verifying the finite reachability.

2. **The outer cycle** ($\dagger_2$): Evaluates the reset and infinite repetition. The right-thread passes through the entire body, meaning both the inner μ-variable U and the outer ν-variable V are unfolded infinitely often ($\text{Inf}(\tau) = \{V, U\}$). Because V structurally binds outside U ($V <_\Phi U$), the outermost infinitely unfolded variable is exactly V. As V is a ν-variable on the right, it forms an admissible ν-thread verifying the infinite safety.

This nested example powerfully illustrates how our purely syntax-driven dependency order ($<_\Phi$) handles complex alternations without requiring explicit invariants.

5 Soundness

To prove soundness, we rely on the concept of *signatures* to explicitly bound the unfoldings of fixed-point variables within the transfinite semantics. This signature-based approach adapts the well-foundedness arguments by Dax et al. [7] to our cyclic proof system, extended to accommodate the closure ordinal κ.



$$\Phi_{down} \equiv \mu X.(p \frown X \vee q) \quad \Phi_{prog} \equiv \nu Y.(\Phi_{down} \frown Sb_x^N \frown Y) \quad \Phi_{inL} \equiv \Phi_{down} \frown Sb_x^N \frown \Phi_{prog}$$
$$\Phi'_{down} \equiv \mu U.(p \frown U \vee q \frown x = 0) \quad \Phi_{spec} \equiv \nu V.(\Phi'_{down} \frown Sb_x^N \frown V) \quad \Phi_{inR} \equiv \Phi'_{down} \frown Sb_x^N \frown \Phi_{spec}$$
$$p \equiv x > 0 \frown Sb_x^{x-1} \quad q \equiv x \leq 0 \frown Id$$

Figure 4: Cyclic proof tree for nested μ/ν alternation (S_{nest}).

Definition 5.1 (μ- and ν-Signatures)**.** To construct a well-founded metric for tracking invalid traces, we extract the sequences of fixed-point variables from the root formula Φ. Let $\mathscr{X}_\Phi^\mu = (X_1^\mu, \ldots, X_n^\mu)$ be the sequence of all μ-variables occurring in Φ, and $\mathscr{X}_\Phi^\nu = (X_1^\nu, \ldots, X_m^\nu)$ be the sequence of all ν-variables in Φ. Both sequences are topologically sorted according to the dependency order $<_\Phi$ ($X_i^\mu <_\Phi X_j^\mu \implies i < j$).

For a left-thread (which tracks formulas in the antecedent), we define a *μ-signature* ξ as a tuple of ordinals $(\alpha_1, \ldots, \alpha_n)$ where each $\alpha_i \leq \kappa$. For a right-thread (which tracks formulas in the succedent), we define a *ν-signature* ξ as a tuple of ordinals $(\alpha_1, \ldots, \alpha_m)$ where each $\alpha_i \leq \kappa$.

A μ-signature or ν-signature $\xi = (\alpha_1, \ldots, \alpha_l)$ for a variable sequence $(X_1, \ldots, X_l)$ can naturally be viewed as an approximation level $\xi : \mathscr{X} \to \kappa + 1$ defined by:

$$\xi(X) = \begin{cases} \alpha_i & \text{if } X = X_i \text{ for some } 1 \leq i \leq l, \\ \kappa & \text{otherwise.} \end{cases}$$

This mapping allows us to evaluate the semantics of trace formulas under specific ordinal approximation levels for the variables relevant to the thread, while leaving all other variables at the maximum bound κ.

We compare these signatures using the standard lexicographic order $\leq_{lex}$, reading from left to right. Since the class of ordinals is well-ordered, the strict lexicographic order $<_{lex}$ on finite tuples of ordinals up to κ is also well-ordered.

In what follows, we only consider formulas Φ without free fixed-point variables. Since their semantics do not depend on a valuation $\mathscr{V}$, we simply write $[\![\Phi]\!]^\xi$ or $[\![\Phi]\!]$.

Definition 5.2 (Falsifying μ- and ν-Signatures)**.** Let Φ be a formula tracked in a thread, and σ be a witness trace. If Φ is in the antecedent and $\sigma \in [\![\Phi]\!]$, the *least falsifying μ-signature* ξ for Φ via σ is defined as the lexicographically least μ-signature such that $\sigma \in [\![\Phi]\!]^\xi$. Conversely, if Φ is in the succedent and $\sigma \notin [\![\Phi]\!]$, its *least falsifying ν-signature* ξ is the lexicographically least ν-signature such that $\sigma \notin [\![\Phi]\!]^\xi$. The existence of such a least signature is guaranteed by the well-ordering of $\leq_{lex}$.

Proposition 5.3 (Local Soundness)**.** Let r be an inference rule applied to an invalid sequent $S_i = \Gamma_i \vdash \Delta_i$ falsified by a trace σ_i. Then, there exists a premise $S_{i+1} = \Gamma_{i+1} \vdash \Delta_{i+1}$ falsified by some trace σ_{i+1}. Furthermore, for any tracked formula Φ_i in S_i and its direct ancestor Φ_{i+1} in S_{i+1}, let ξ_i and ξ_{i+1} be their least falsifying μ-signatures (if tracking on the left) or ν-signatures (if tracking on the right). We have:

- $\xi_{i+1} \leq_{lex} \xi_i$.

- If r is the fixed-point unfolding rule applied to Φ_i (i.e., unfolding $\mu X.\Phi$ on the left or $\nu X.\Phi$ on the right), and no fixed-point variable strictly outside X is unfolded, then $\xi_{i+1} <_{lex} \xi_i$.

Proof. We proceed by case analysis on the inference rule r.

Fixed-Point Unfoldings

For the fixed-point unfolding rules ((UNFL), (CH-UNFL), (UNFR), and (CH-UNFR)), as established in [7, 8], the signature strictly decreases. Let X_j be the unfolded tracked variable. Suppose the least falsifying signature for the conclusion is $\xi_i = (\alpha_1, \ldots, \alpha_j, \ldots, \alpha_l)$. Assuming left-unfolding of $\mu X_j.\Psi$ (the ν case is symmetric), the semantics $\sigma \in [\![\mu X_j.\Psi]\!]^{\xi_i}$ requires an approximation bound α_j. By the definition of transfinite approximations, whether α_j is a successor or limit ordinal, there always exists a strictly smaller ordinal $\beta < \alpha_j$ such that $\sigma \in [\![\Psi]\!]^{\xi_i[X_j \mapsto \beta]}$. Let $\xi_{i+1} = (\alpha'_1, \ldots, \alpha'_j, \ldots, \alpha'_l)$ be the least signature for the premise tracking $\Psi[\mu X_j.\Psi/X_j]$. Since bound variables are distinct and ordered by $<_\Phi$, the substitution preserves the bounds of all outer variables: $\alpha'_m = \alpha_m$ for all $m < j$. For X_j itself, the required bound strictly decreases: $\alpha'_j \leq \beta < \alpha_j$. Although inner bounds (α'_m for $m > j$) may change arbitrarily, the identical prefix and the strict decrease at index j entirely determine the lexicographic order, yielding $(\alpha_1, \ldots, \alpha'_j, \ldots) <_{lex} (\alpha_1, \ldots, \alpha_j, \ldots)$. Thus, $\xi_{i+1} <_{lex} \xi_i$.

Other Rules

For all other logical and structural rules, we demonstrate that the least falsifying μ- or ν-signature is exactly preserved (i.e., $\xi_{i+1} = \xi_i$), which trivially satisfies the condition $\xi_{i+1} \leq_{lex} \xi_i$. Due to space limitations, we only detail the proofs for the representative rules: (CUT), (REL), (CH-UPD), and (ARB2). The proofs for the remaining rules follow analogously. Let ξ_i be the least falsifying signature for the invalid conclusion via the witness trace σ.

(CUT): Assume the conclusion $\Gamma \vdash \Delta$ is falsified by σ under ξ_i. By the law of excluded middle in the state semantics, either $\sigma \in [\![p]\!]^{\xi_i}$ or $\sigma \notin [\![p]\!]^{\xi_i}$. If $\sigma \in [\![p]\!]^{\xi_i}$, then $\sigma \in [\![\bigwedge \Gamma \wedge p]\!]^{\xi_i}$, making the left premise $\Gamma, p \vdash \Delta$ falsified by σ. If $\sigma \notin [\![p]\!]^{\xi_i}$ (i.e., $\sigma \in [\![\bar{p}]\!]^{\xi_i}$), the right premise $\Gamma, \bar{p} \vdash \Delta$ is falsified by σ. In both cases, $\xi_{i+1} = \xi_i$.

(REL): Assume the conclusion $\Gamma, R \vdash R', \Delta$ is falsified by σ. This implies $\sigma \in [\![\bigwedge \Gamma \wedge R]\!]^{\xi_i}$ and $\sigma \notin [\![R' \vee \bigvee \Delta]\!]^{\xi_i}$. However, $\sigma \in [\![\bigwedge \Gamma]\!]^{\xi_i}$ implies that the first state of σ satisfies P_Γ. Thus, $\sigma \in [\![P_\Gamma \cap R]\!]^{\xi_i}$. By the rule's side condition $R|_{P_\Gamma} \subseteq R'$, it must hold that $\sigma \in [\![R']\!]^{\xi_i}$, which contradicts $\sigma \notin [\![R']\!]^{\xi_i}$. Therefore, this rule can never be applied to an invalid sequent.

(CH-UPD): Assume the conclusion $\Gamma, Sb^a_x \frown \Phi \vdash \ldots$ is falsified by σ under ξ_i. The semantics of the sequential composition (chop) and the substitution relation Sb^a_x dictate how the trace σ transitions. Since trace decomposition merely advances the state evaluation without consuming fixed-point unfoldings, the signature remains constant: $\xi_{i+1} = \xi_i$.

(ARB2): Assume the conclusion $\Gamma, \Phi_1 \frown \Phi_2 \vdash \mathit{true} \frown \Psi, \Delta$ is falsified by σ under ξ_i. This implies $\sigma \in [\![\Phi_1 \frown \Phi_2]\!]^{\xi_i}$ and $\sigma \notin [\![\mathit{true} \frown \Psi \vee \bigvee \Delta]\!]^{\xi_i}$. By the associative semantics of the chop operator, we have the semantic inclusion $[\![\Phi_1 \frown \mathit{true} \frown \Psi]\!]^{\xi_i} \subseteq [\![\mathit{true} \frown \Psi]\!]^{\xi_i}$. Because $\sigma \notin [\![\mathit{true} \frown \Psi]\!]^{\xi_i}$, it must be that $\sigma \notin [\![\Phi_1 \frown \mathit{true} \frown \Psi]\!]^{\xi_i}$. Thus, the premise is falsified by σ under $\xi_{i+1} = \xi_i$. □

Theorem 5.4 (Soundness)**.** If there exists a cyclic proof $(\mathscr{D}, F)$ of a sequent $\Gamma \vdash \Delta$, then the sequent is valid.

Proof. We proceed by contradiction. Suppose there exists a cyclic proof $(\mathscr{D}, F)$ of an invalid root sequent $S_0 = \Gamma \vdash \Delta$.

By repeatedly applying Proposition 5.3, we construct an infinite path $\pi = (S_i)_{i \geq 0}$ of invalid sequents, where each S_i is falsified by a witness trace σ_i. Since $(\mathscr{D}, F)$ is a cyclic proof, there must exist an admissible

thread $\tau = (\Phi_i)_{i\geq 0}$ along π. According to Definition 3.5, let X_{out} be the outermost infinitely unfolded variable in τ.

For each $i \geq 0$, let ξ_i be the least falsifying μ-signature (if τ is a left-thread) or ν-signature (if τ is a right-thread) of Φ_i via σ_i. By Proposition 5.3, the sequence of these signatures is monotonically non-increasing under the lexicographic order:

$$\xi_0 \geq_{lex} \xi_1 \geq_{lex} \xi_2 \geq_{lex} \cdots .$$

By the definition of an admissible thread, any variable Y that binds strictly outside X_{out} ($Y <_\Phi X_{out}$) is unfolded only finitely many times in τ. Therefore, there exists an index N such that for all $i \geq N$, no variable strictly outside of X_{out} is ever unfolded.

From index N onward, the signature components $\xi_i(Y)$ for all $Y <_\Phi X_{out}$ remain constant. As established in Proposition 5.3, the distinctness and structural nesting of variables guarantee that the repeated unfoldings of X_{out} and its inner variables do not affect the ordinal bounds of these outer variables.

However, because the thread τ is admissible, X_{out} is a μ-variable on the left or a ν-variable on the right, and it is unfolded infinitely often. When X_{out} is unfolded at some step $k \geq N$, the condition for strict decrease in Proposition 5.3 holds (i.e., no variable strictly outside X_{out} is unfolded). Thus, we obtain a strict lexicographic decrease $\xi_{k+1} <_{lex} \xi_k$, which contradicts the well-foundedness $<_{lex}$.

Thus, the initial assumption must be false, and the root sequent S_0 is valid. □

6 Conclusion

In this paper, we proposed a cyclic proof system for verifying trace formula implications. To specify both finite and non-terminating executions, we extended trace logic with the greatest fixed-point operator ν. Our system represents inductive and coinductive reasoning by proof cycles, reducing the need for explicit invariants. We defined admissible threads for both μ- and ν-variables and proved soundness using lexicographically ordered falsifying signatures.

Future Work. Several directions remain:

- **Completeness and Automation.** Completeness of the calculus remains open. A future prover could combine systematic rule application, SMT solving for state predicates, backlink generation, and an Infinite Descent checker such as Cyclone [6]. Identifying useful decidable fragments is also important for automated proof search.
- **Trace Formula Characterization for Infinite Behaviors.** For finite executions, $stf(S)$ precisely characterizes the program traces. For maximal executions containing both finite and infinite traces, an exact compositional translation may require suitable combinations of μ and ν, rather than uniformly replacing one by the other.
- **Extension to Mutual and Non-Tail Recursion.** While Heidler et al. [10] use method contracts for general recursive programs, our current system mainly handles recursion by direct fixed-point unfolding. Extending the cyclic rules to mutual and non-tail recursion is therefore an important direction.

References

[1] J. Brotherston & N. Gorogiannis (2014): *Cyclic abduction of inductively defined safety and termination preconditions*. In: *SAS*, pp. 68–84, doi:10.1007/978-3-319-10936-7_5.

[2] J. Brotherston, N. Gorogiannis & R.L. Petersen (2012): *A generic cyclic theorem prover*. In: *APLAS*, pp. 350–367, doi:10.1007/978-3-642-35182-2_25.

[3] J. Brotherston & A. Simpson (2011): *Sequent calculi for induction and infinite descent*. *J. Log. Comput.* 21(6), pp. 1177–1216, doi:10.1093/logcom/exq052.

[4] Xiaohong Chen, Dorel Lucanu & Grigore Rosu (2021): *Matching logic explained*. *J. Log. Algebraic Methods Program.* 120, p. 100638, doi:10.1016/J.JLAMP.2021.100638. Available at `https://doi.org/10.1016/j.jlamp.2021.100638`.

[5] Liron Cohen & Reuben N. S. Rowe (2020): *Integrating Induction and Coinduction via Closure Operators and Proof Cycles*. In Nicolas Peltier & Viorica Sofronie-Stokkermans, editors: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, *Lecture Notes in Computer Science* 12166, Springer, pp. 375–394, doi:10.1007/978-3-030-51074-9_21. Available at `https://doi.org/10.1007/978-3-030-51074-9_21`.

[6] Liron Cohen, Reuben N. S. Rowe & Matan Shaked (2025): *Cyclone: A Heterogeneous Tool for Verifying Infinite Descent*. In Arie Gurfinkel & Marijn Heule, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part I*, *Lecture Notes in Computer Science* 15696, Springer, pp. 336–354, doi:10.1007/978-3-031-90643-5_18. Available at `https://doi.org/10.1007/978-3-031-90643-5_18`.

[7] C. Dax, M. Hofmann & M. Lange (2006): *A Proof System for the Linear Time μ-Calculus*. In: *FSTTCS*, LNCS, pp. 273–284, doi:10.1007/11944836_26.

[8] A. Doumane (2017): *On the infinitary proof theory of logics with fixed points*. Ph.D. thesis, Paris Diderot University, France. Available at `https://tel.archives-ouvertes.fr/tel-01676953`.

[9] D. Gurov & R. Hähnle (2025): *An Expressive Trace Logic for Recursive Programs*. In: *FSCD*, *LIPIcs* 337, pp. 21:1–21:22, doi:10.4230/LIPICS.FSCD.2025.21.

[10] N. Heidler & R. Hähnle (2025): *A Sequent Calculus For Implication*. In: *TABLEAUX*, *LNCS* 15980, pp. 473–490, doi:10.1007/978-3-032-06085-3_25.

[11] C.A.R. Hoare (1969): *An axiomatic basis for computer programming*. *Commun. ACM* 12(10), pp. 576–580, doi:10.1145/363235.363259.

[12] Dorel Lucanu, Vlad Rusu & Andrei Arusoaie (2017): *A generic framework for symbolic execution: A coinductive approach*. *J. Symb. Comput.* 80, pp. 125–163, doi:10.1016/J.JSC.2016.07.012. Available at `https://doi.org/10.1016/j.jsc.2016.07.012`.

[13] C.-H. L. Ong (2015): *Automata, Logic and Games*. `http://logica.dmi.unisa.it/tacl/wp-content/uploads/2015/05/AutomataLogicGames2015.pdf`. Lecture notes.

[14] R.N.S. Rowe & J. Brotherston (2017): *Automatic cyclic termination proofs for recursive procedures in separation logic*. In: *CPP*, pp. 53–65, doi:10.1145/3018610.3018623.

[15] Andrei Stefanescu, Ştefan Ciobâcă, Radu Mereuta, Brandon M. Moore, Traian-Florin Serbanuta & Grigore Rosu (2019): *All-Path Reachability Logic*. *Log. Methods Comput. Sci.* 15(2), doi:10.23638/LMCS-15(2:5)2019. Available at `https://doi.org/10.23638/LMCS-15(2:5)2019`.

On Comparing Python Programs Based on Differences in Rewrite Sequences to Support Grading Programming Exercises *

Misaki Kojima
Nagoya University, Nagoya, Japan
kojima@i.nagoya-u.ac.jp

Naoki Nishida
Nagoya University, Nagoya, Japan
nishida@i.nagoya-u.ac.jp

In an introductory programming class using, e.g., Python, students are often asked to modify sample programs under specific coding constraints so that the modified programs print expected results using print statements in the sample programs. In grading the submitted programs, checking only the correctness of the required printing is insufficient, as it cannot detect unacceptable computational behaviors, deceptive printing, and/or violations of coding constraints. In this paper, we propose a method to support the grading of introductory programming assignments based on the comparison of rewrite sequences of rewrite systems obtained from a pre-prepared answer program and students' submitted programs. In our method, both the answer program and submitted programs are transformed into logically constrained rewrite systems, respectively, and their rewrite sequences from the same initial state for the required execution of the assignment are compared. This enables us to detect differences in execution, such as the use of different constructs or incorrect loop conditions.

1 Introduction

Recently, approaches to program verification by means of logically constrained term rewrite systems (LCTRSs, for short) [9] have been extensively studied [6, 5, 13, 7, 4, 11, 8, 12]. LCTRSs are useful models of not only functional but also imperative programs (cf. [6]). For instance, equivalence checking by means of LCTRSs is useful to ensure correctness of terminating functions. Here, equivalence of two functions means that for every input, the functions return the same output or end with the same projection of final configurations.

In general, equivalence of functions does not consider the computation process from inputs to outputs, and conventional *equivalence checking* primarily focuses on the I/O relationships between two programs or functions, e.g., equivalence of $\{(v_1,\ldots,v_n,v_o) \mid f(v_1,\ldots,v_n) = v_o\}$ and $\{(v'_1,\ldots,v'_n,v'_o) \mid g(v'_1,\ldots,v'_n) = v'_o\}$ for n-ary functions f,g. Equivalence checking is a useful formal method, but it is often computationally expensive, while transforming programs into, e.g., LCTRSs (cf. [6]), is generally very efficient. For example, equivalence-checking tools based on *rewriting induction* (RI, for short) [15, 6] first prove termination of given rewrite systems such as term rewrite systems (TRSs, for short) and LCTRSs; then, they usually prove *quasi-reducibility* (non-existence of undefined patterns); finally, they start the inference process for given (constrained) equations, where *lemma generation* is usually used for automation (cf. [6, Section 5]). Despite the inefficiency, formal methods for equivalence verification are very useful to ensure the reliability of software such as automotive embedded systems that are critical to human life.

Despite the importance mentioned above, formal methods are sometimes not only too expensive but also insufficient to support human tasks. In an introductory programming class using, e.g., Python, some

*This work was partially supported by JSPS KAKENHI Grant Number 24K02900.

assignments already include sample programs and require students to modify them under certain coding constraints so that the modified programs print expected results using `print` statements in the sample programs. It is usual to prepare a model answer program in advance, and thus a submitted program is correct if it is equivalent to the answer program. The use of formal methods to verify equivalence of two programs is adequate and makes the grading task accurate. However, when applying a formal method to verify equivalence of the two programs for introductory programming assignments, the limitation of focusing only on final results is insufficient for grading assignments from the educational viewpoint. For example, even if the output is correct, there can be unacceptable internal behaviors, errors invisible from the output, and/or syntactic violations, regarding the computation process or compliance with the assignment specification such as coding constraints. Such cases include situations where the loop termination condition does not match the intended specification or where the required program structure is not used.

Figure 1 shows a part of an actual assignment of a first-year undergraduate class for Python programming exercises at Nagoya University, where approximately 1,300 students take that class each year, and about 20 teaching assistants check the correctness of submitted programs for the grading. The assignment requires students to write a Python program that calculates an approximate value of π using Archimedes' method. The objective of the assignment is to study a `while` loop, and thus requires students to use a `while` loop and to use N in the loop condition. For grading, the automation of checking correctness of submitted programs can reduce the burden on humans and improve the accuracy of the work. On the other hand, formal methods based on, e.g., RI, are excessive for simple tasks such as the grading of small programs that are generated from sample programs. In addition, the application of expensive methods to 1,300 submitted programs is not reasonable.

The Python program P_1 of Listing 1 is a model answer for the assignment in Figure 1, which is prepared by a lecturer in advance. Most students write programs that are syntactically equivalent to the answer one. String-level differences using, e.g., the `diff` command on plain texts are more efficient than checking string-level equivalence by eye. However, such differences are not always effective because most of such programs may not perfectly match the answer at the string level, due to the inclusion of, e.g., extra blanks and line breaks. For the same reason, the command line option "`-m trace --trace`", which enables us to observe exactly what lines are executing, is not complete. In addition, some may write `n<=3071` or `3072>n` instead of `n<3072`, and a program in which the `print` statement is indented and executed at each iteration of the loop (Listing 2), denoted by P_2, is also considered correct: The indent on line 9 must be inserted to confirm intermediate results for debugging.

On the other hand, we would like to automatically detect incorrect programs as much as possible and even identify what is wrong with them. Regarding the assignment in Figure 1, the following incorrect programs have sometimes been submitted: A program of Listing 3, denoted by P_3, that uses "`for i in range(9)`" instead of the required `while` statement, and a program of Listing 4, denoted by P_4, that uses a `while` loop with an incorrect loop condition "`n <= 3072`". The first incorrect program P_3 produces the same output as the correct program, but it does not satisfy the coding requirement "by using a `while` loop". The second incorrect program P_4 satisfies the coding requirement and computes the range of π when $N = 3072$ during execution; however, it performs one extra iteration of the loop, resulting in a different final output. Such differences cannot be detected by either formal methods that only check the equivalence of the I/O relationships or by approaches based on the reachability of expected outputs.

In this paper, we propose a method to support the grading of introductory programming assignments based on the comparison of rewrite sequences of LCTRSs obtained from a pre-prepared answer program and students' submitted programs. In our method, both the answer program and submitted programs are transformed into LCTRSs, respectively (Section 4), and their rewrite sequences from the same initial state for the required execution of the assignment are compared (Section 5). Since Python programs of

Let p_N be half the perimeter of a regular N-gon inscribed in a circle of radius 1, and let q_N be half the perimeter of a regular N-gon circumscribed around the circle. Then, the following holds:

$$p_N < \pi < q_N$$

Furthermore, let p_{2N} and q_{2N} be the half-perimeters of the regular $2N$-gon inscribed in a circle of radius 1 and the regular $2N$-gon circumscribed around the circle, respectively. Then the following holds:

$$q_{2N} = \frac{2p_Nq_N}{p_N+q_N} \qquad p_{2N} = \sqrt{p_Nq_{2N}}$$

When $N = 6$, we have $p = 3$ and $q = 2\sqrt{3}$. Using this, the following program computes the range of π when $N = 12$.

```
1 p = 3
2 q = 2 * 3 ** 0.5
3 n = 6
4
5 n = n * 2
6 q = 2 * p * q / (p + q)
7 p = (p*q)**0.5
8 print('{0}-gon: π is between {1:.6f} and {2:.6f}'.format(n,p,q))
```

Modify this program to compute the range of π when $N = 3072$ by using a `while` loop, and by using N in the loop condition.

Figure 1: An assignment in an introductory Python programming class

the assignment in Figure 1 deal with floating point numbers, we define a theory signature for floating point numbers (Section 3).

By using rewrite sequences, it is possible to capture not only the equivalence of final outputs but also differences in computation processes. For example, the difference between a `while` loop and a `for` loop can be identified as a difference in the function symbols appearing in the sequences, and an extra iteration caused by an incorrect loop condition appears as a difference in the length of the sequences. On the other hand, non-essential differences such as the location of `print` statements or debugging outputs also affect the sequences. In comparing rewrite sequences, we ignore such differences by removing the corresponding terms from the sequences, while we do not remove the required `print` statement for the final output. Furthermore, by preparing multiple answer programs and their sequences, the method can flexibly handle different implementations that can be considered correct.

One may think that equivalence of transformed LCTRSs also works for the grading. To check the equivalence, we have to check equivalence of constrained rewrite rules $\ell \to r\ [\phi]$ and $\ell \to r\ [\phi']$, which may have the same left- and right-hand sides. For such equivalence, we have to verify validity of $\phi \Leftrightarrow \phi'$ by calling an SMT solver. On the other hand, to rewrite a ground term by a constrained rewrite rule $\ell \to r\ [\phi]$, we evaluate a quantifier-free closed constraint $\phi\theta$, where θ is a matching substitution. The evaluation is easy and we do not have to call any SMT solver. In addition, rewrite sequences can be efficiently compared using an interpreter such as Ctrl [10] because they can be printed as plain texts under a fixed format. For this reason, checking equivalence of finite rewrite sequences is, in general,

Listing 1: Correct program P_1 using a `while` loop

```
p = 3
q = 2 * 3 ** 0.5
n = 6

while(n<3072):
  n = n * 2
  q = 2 * p * q / (p + q)
  p = (p*q)**0.5
print('{0}-gon: π is between {1:.6f} and {2:.6f}'.format(n,p,q))
```

Listing 2: A variant P_2 of the correct program P_1

```
p = 3
q = 2 * 3 ** 0.5
n = 6

while(n<3072):
  n = n * 2
  q = 2 * p * q / (p + q)
  p = (p*q)**0.5
  print('{0}-gon: π is between {1:.6f} and {2:.6f}'.format(n,p,q))
```

more efficient than checking equivalence of LCTRSs.

2 Preliminaries

In this section, we briefly recall LCTRSs [9, 6]. Familiarity with basic notions and notations on term rewriting is assumed [1, 14].

To define an LCTRS over an $\mathcal{S}$-sorted signature Σ, we consider the following sorts, signatures, mappings, and constants: *Theory sorts* in $\mathcal{S}_{theory}$ and *term sorts* in $\mathcal{S}_{term}$ such that $\mathcal{S} = \mathcal{S}_{theory} \uplus \mathcal{S}_{term}$; a *theory signature* Σ_{theory} and a *term signature* Σ_{terms} such that $\Sigma = \Sigma_{theory} \cup \Sigma_{terms}$ and $\iota_1, \ldots, \iota_n, \iota \in \mathcal{S}_{theory}$ for any symbol $f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota \in \Sigma_{theory}$; a mapping $\mathcal{I}$ that assigns to each theory sort ι a (non-empty) set $\mathcal{A}_\iota$, so-called the universe of ι (i.e., $\mathcal{I}(\iota) = \mathcal{A}_\iota$); a mapping $\mathcal{J}$, so-called an interpretation for Σ_{theory}, that assigns to each function symbol $f : \iota_1 \times \cdots \times \iota_n \Rightarrow \iota \in \Sigma_{theory}$ a function $f^{\mathcal{J}}$ in $\mathcal{I}(\iota_1) \times \cdots \times \mathcal{I}(\iota_n) \to \mathcal{I}(\iota)$ (i.e., $\mathcal{J}(f) = f^{\mathcal{J}}$); a set $\mathcal{V}al_\iota \subseteq \Sigma_{theory}$ of *value-constants* $a : \iota$ for each theory sort ι such that $\mathcal{J}$ gives a bijection from $\mathcal{V}al_\iota$ to $\mathcal{I}(\iota)$ $(= \mathcal{A}_\iota)$. We denote $\bigcup_{\iota \in \mathcal{S}_{theory}} \mathcal{V}al_\iota$ by $\mathcal{V}al$. Note that $\mathcal{V}al \subseteq \Sigma_{theory}$. For readability, we may not distinguish $\mathcal{V}al_\iota$ and $\mathcal{I}(\iota)$ $(= \mathcal{A}_\iota)$, i.e., for each $v \in \mathcal{V}al_\iota$, v and $\mathcal{J}(v)$ may be identified. We require that $\Sigma_{terms} \cap \Sigma_{theory} \subseteq \mathcal{V}al$. Symbols in $\Sigma_{theory} \setminus \mathcal{V}al$ are *calculation symbols*, for which we may use infix notation. A term in $T(\Sigma_{theory}, \mathcal{V})$ is called a *theory term*, where $\mathcal{V}$ is a countably infinite set of variables. We define the *interpretation* $[\![\cdot]\!]_{\mathcal{J}}$ of ground theory terms as $[\![f(s_1, \ldots, s_n)]\!]_{\mathcal{J}} = \mathcal{J}(f)([\![s_1]\!]_{\mathcal{J}}, \ldots, [\![s_n]\!]_{\mathcal{J}})$. Note that for every ground theory term s, there is a unique value-constant c such that $[\![s]\!]_{\mathcal{J}} = [\![c]\!]_{\mathcal{J}}$.

We typically choose a theory signature $\mathcal{S}_{theory}$ such that $\mathcal{S}_{theory} \supseteq \mathcal{S}_{core} = \{bool\}$, $\mathcal{V}al_{bool} = \{\mathsf{true}, \mathsf{false}\}$, $\Sigma_{theory} \supseteq \Sigma_{core} = \mathcal{V}al_{bool} \cup \{\wedge, \vee, \Rightarrow, \Leftrightarrow : bool \times bool \Rightarrow bool,\ \neg : bool \Rightarrow bool\} \cup \{=_\iota, \neq_\iota : \iota \times \iota \Rightarrow$

Listing 3: An unexpected variant P_3 of P_1

```
p = 3
q = 2 * 3 ** 0.5
n = 6

for i in range(9):
  n = n * 2
  q = 2 * p * q / (p + q)
  p = (p*q)**0.5
print('{0}-gon: π is between {1:.6f} and {2:.6f}'.format(n,p,q))
```

Listing 4: An incorrect variant P_4 of P_1

```
p = 3
q = 2 * 3 ** 0.5
n = 6

while(n<=3072):
  n = n * 2
  q = 2 * p * q / (p + q)
  p = (p*q)**0.5
print('{0}-gon: π is between {1:.6f} and {2:.6f}'.format(n,p,q))
```

$bool \mid \iota \in \mathcal{S}_{theory}\}$, $\mathcal{I}(bool) = \{\mathsf{true}, \mathsf{false}\}$, and $\mathcal{J}$ interprets these symbols as expected. We omit the sort subscripts ι from $=_\iota$ and $\neq_\iota$ when they are clear from the context. A theory term with sort *bool* is called a *constraint*. A substitution γ which is a sort-preserving mapping from $T(\Sigma,\mathcal{V})$ to $T(\Sigma,\mathcal{V})$ is said to *respect* a constraint ϕ if $x\gamma \in \mathcal{V}al$ for all $x \in \mathcal{V}ar(\phi)$ and $[\![\phi\gamma]\!]_{\mathcal{J}} = \mathsf{true}$, where $\mathcal{V}ar(\phi)$ denotes the set of variables appearing in ϕ.

A *constrained rewrite rule* is a triple $\ell \to r\ [\varphi]$ such that ℓ and r are terms of the same sort, φ is a constraint, and ℓ is not a theory term (and thus, ℓ is not a variable). If $\varphi = \mathsf{true}$, then we may write $\ell \to r$. We define $\mathcal{LV}ar(\ell \to r\ [\varphi])$ as $\mathcal{V}ar(\varphi) \cup (\mathcal{V}ar(r) \setminus \mathcal{V}ar(\ell))$, the set of *logical variables in* $\ell \to r\ [\varphi]$ which are variables instantiated with values in rewriting terms. We say that a substitution γ *respects* $\ell \to r\ [\varphi]$ if $\gamma(x) \in \mathcal{V}al$ for all $x \in \mathcal{LV}ar(\ell \to r\ [\varphi])$ and $[\![\varphi\gamma]\!]_{\mathcal{J}} = \mathsf{true}$. Regarding the signature of $\mathcal{R}$, we denote the set $\{f(x_1,\ldots,x_n) \to y\ [y = f(x_1,\ldots,x_n)] \mid f \in \Sigma_{theory} \setminus \mathcal{V}al,\ x_1,\ldots,x_n,y \in \mathcal{V} \text{ are pairwise distinct}\}$ by $\mathcal{R}_{calc}$. The elements of $\mathcal{R}_{calc}$ are *calculation rules* and we often deal with them as constrained rewrite rules even though their left-hand sides are theory terms. The *rewrite relation* $\to_{\mathcal{R}}$ is a binary relation over terms, defined as follows: For a term s, $s[\ell\gamma]_p \to_{\mathcal{R}} s[r\gamma]_p$ if and only if $\ell \to r\ [\varphi] \in \mathcal{R} \cup \mathcal{R}_{calc}$ and γ respects $\ell \to r\ [\varphi]$. A reduction step with $\mathcal{R}_{calc}$ is called a *calculation*. A *logically constrained term rewrite system* (LCTRS, for short) is defined as an abstract reduction system $(T(\Sigma,\mathcal{V}), \to_{\mathcal{R}})$, simply denoted by $\mathcal{R}$, where $\mathcal{R}$ is a set of constrained rewrite rules. An LCTRS is usually given by supplying Σ, $\mathcal{R}$, and an informal description of $\mathcal{I}$ and $\mathcal{J}$ if these are not clear from the context.

The *integer signature* Σ_{int} is $\Sigma_{core} \cup \{+,-,\times,\mathsf{exp},\mathsf{div},\mathsf{mod} : int \times int \Rightarrow int\} \cup \{\geq, > : int \times int \Rightarrow bool\} \cup \mathcal{V}al_{int}$ where $\mathcal{S}_{theory} \supseteq \{int, bool\}$, $\mathcal{V}al_{int} = \mathbb{Z}$, and $\mathcal{I}(int) = \mathbb{Z}$. We define $\mathcal{J}$ based on the SMT-Lib semantics [2]. An LCTRS over a signature $\Sigma \supseteq \Sigma_{int}$ with $\Sigma_{theory} = \Sigma_{int}$ is called an *integer LCTRS*.

Example 2.1 The following integer LCTRS calculates the *factorial* function over $\mathbb{Z}$ iteratively:

$$\mathcal{R}_1 = \left\{ \begin{array}{ll} \mathsf{fact}(x) \to \mathsf{subfact}(x,1) & \\ \mathsf{subfact}(x,y) \to y & [x \leq 0] \\ \mathsf{subfact}(x,y) \to \mathsf{subfact}(x-1, x \times y) & [x > 0] \end{array} \right\}$$

The term $\mathsf{fact}(3)$ is reduced by $\mathcal{R}_1$ to 6: $\mathsf{fact}(3) \to_{\mathcal{R}_1} \mathsf{subfact}(3,1) \to_{\mathcal{R}_1} \mathsf{subfact}(3-1, 3\times 1) \to_{\mathcal{R}_1} \mathsf{subfact}(2, 3\times 1) \to_{\mathcal{R}_1} \mathsf{subfact}(2-1, 2\times(3\times 1)) \to_{\mathcal{R}_1} \mathsf{subfact}(1, 2\times(3\times 1)) \to_{\mathcal{R}_1} \mathsf{subfact}(1-1, 1\times(2\times(3\times 1))) \to_{\mathcal{R}_1} \mathsf{subfact}(0, 1\times(2\times(3\times 1))) \to_{\mathcal{R}_1} 1\times(2\times(3\times 1)) \to^*_{\mathcal{R}_1} 6$.

3 LCTRSs with Theories of Integers and Floating Point Numbers

Python programs in Section 1 include not only integers but also floating point numbers. To transform such programs into LCTRSs, in this section, we prepare a signature for the theory of floating point numbers [16, 3]. To simplify discussions, we deal with 32-bit floating-point numbers only. The signature we define below follows the theory of floating point numbers in the SMT-LIB [2], where floating point numbers are represented as triples of *bit-vectors*.

A (fixed-size) n-bit *bit-vector* is a sequence of $0, 1$ of length n. The set of n-bit bit-vectors is denoted by $\mathbb{BV}_n$. Following the SMT-LIB notation, a bit-vector $c \in \mathbb{BV}_n$ is denoted by $\#\mathsf{b}c$. When n is a multiple of four, we often use hexadecimal notations $\#\mathsf{x}c'$ for a binary numeral $c \in \mathbb{BV}_n$, where c' is the corresponding hexadecimal digit of c.

Python's standard floating-point numbers with type `float` follow the IEEE 754 standard[1] for double-precision floating-point format (64-bit), which is equivalent to the `double` type in many other programming languages. Note that the theory `FloatingPoint` of the SMT-LIB also follows the IEEE 754 standard and the sort `Float64` in the SMT-LIB corresponds to `float` of Python. For this reason, we use floating-point numbers based on the IEEE 754 standard.

Floating-point numbers are represented by 32-bit bit-vectors such that 64-bit space is split into a sign bit (1 bit), an exponent (11 bits), and a significand/fraction (52 bits), which are represented by bit-vectors. Note that `Float64` of the SMT-LIB splits 64-bit space into an exponent (12 bits) and a significand/fraction (53 bits). For a floating-point number with a sign bit $b \in \mathbb{BV}_1$, an exponent $b_e \in \mathbb{BV}_{11}$, and a significand/fraction $b_s \in \mathbb{BV}_{52}$, we use the triple $\langle b, b_e, b_s \rangle_{\mathit{float}}$ for the floating-point number.

Python uses *Round Nearest Ties to Even* (RNE)[2] for floating point numbers by default. Thus, for the semantics of operators over floating point numbers, this paper only uses RNE and omits it from operators. To simplify discussions, we do not use bit-vectors and their operators in the signature for floating point numbers, while the theory of floating point numbers in the SMT-LIB includes bit-vectors.

For the theory of floating point numbers, we prepare the following theory sorts, values, and signature:

- $\mathcal{S}_{\mathit{float}} = \mathcal{S}_{\mathit{core}} \cup \{\mathit{float}\}$,
- $\mathcal{V}al_{\mathit{float}} = \mathcal{V}al_{\mathit{bool}} \cup \{\langle b, b_e, b_s \rangle_{\mathit{float}} : \mathit{float} \mid b \in \mathbb{BV}_1, b_e \in \mathbb{BV}_{11}, b_s \in \mathbb{BV}_{52}\} \cup \{\infty, -\infty, \mathsf{nan}\}$, and
- $\Sigma_{\mathit{float}} = \{+., -., *., /., \mathsf{rem.}, \mathsf{max.}, \mathsf{min.} : \mathit{float} \times \mathit{float} \Rightarrow \mathit{float}\} \cup \{\mathsf{abs}, \mathsf{neg}, \mathsf{fma.}, \sqrt{\cdot}, : \mathit{float} \Rightarrow \mathit{float}\} \cup \{\leq., <., \geq., >., =., \neq., : \mathit{float} \times \mathit{float} \Rightarrow \mathit{bool}\}$,

Note that the signature above does not fully cover the theory of floating point numbers in the SMT-LIB. Floating point numbers $\langle b, b_e, b_s \rangle_{\mathit{float}}$ are considered constants in the term setting, while they are triples. The interpretation for Σ_{float} follows that in the SMT-LIB. For readability, we use decimal notations for floating point numbers, e.g., 2.5 stands for, e.g., $\langle \#\mathsf{b}0, \#\mathsf{b}10000000000, \#\mathsf{x}40000000000000 \rangle_{\mathit{float}}$.

[1] `http://grouper.ieee.org/groups/754/`

[2] `roundNearestTiesToEven` (`RNE`) in the SMT-LIB.

$$\mathcal{R}_1 = \left\{ \begin{array}{rll} \mathsf{assign}_1 \to \mathsf{assign}_2(3.0) & & \\ \mathsf{assign}_2(p) \to \mathsf{assign}_3(p,q) & [\, q = 2.0 *. \sqrt{3.0}\,] \\ \mathsf{assign}_3(p,q) \to \mathsf{while}_1(p,q,6) & & \\ \mathsf{while}_1(p,q,n) \to \mathsf{assign}_4(p,q,n) & [\, n < 3072\,] \\ \mathsf{while}_1(p,q,n) \to \mathsf{print}_1(p,q,n) & [\, \neg(n < 3072)\,] \\ \mathsf{assign}_4(p,q,n) \to \mathsf{assign}_5(p,q,n') & [\, n' = n * 2\,] \\ \mathsf{assign}_5(p,q,n) \to \mathsf{assign}_6(p,q',n) & [\, q' = 2.0 *. p *. q /. (p +. q)\,] \\ \mathsf{assign}_6(p,q,n) \to \mathsf{while}_1(p',q,n) & [\, p' = \sqrt{p *. q}\,] \\ \mathsf{print}_1(p,q,n) \to \mathsf{nop}(p,q,n) & & \end{array} \right\}$$

Figure 2: An LCTRS for the program P_1

4 Transformation of Python Programs into LCTRSs

In this section, we show how to transform Python programs into LCTRSs. Note that our target Python programs are sequences of statements.

We adopt a transformation based on existing approaches, while introducing a design that enables the comparison of rewrite sequences. To be more precise, we follow the transformation in [6], while adapting it to Python programs that can be considered simple imperative programs with integers and floating point numbers. The main difference of ours from the original transformation is how to prepare function symbols.

A program state is represented as a term consisting of a function symbol representing the current execution position, and its arguments, which are the values of program variables at that point. Rewrite rules represent the transition between such states. Each statement is transformed into a transition from one execution position to another, with variable values updated as necessary. To avoid the effect of syntactic differences on the comparison of rewrite sequences, for each statement, we generate a function symbol by adding the order in which the same kind of statements appear to the statement name, such as if, rather than the line number in the source code. Since our target program is a sequence of statements, the initial state of the program is represented by the constant introduced for the first statement of the sequence.

Example 4.1 We transform the program P_1 into the LCTRS $\mathcal{R}_1$ with the theories of integers and floating point numbers as illustrated in Figure 2. Each rule in $\mathcal{R}_1$ corresponds directly to a statement in P_1. We prepare the function symbol assign_1 that represents the execution position just before the first assignment `p = 3`. The function symbol plays the role of the initial state and is unary because no variable is declared when starting the execution. The `while` statement is represented by two rules corresponding to the continuation and termination of the loop. When the loop condition no longer holds, the reduction proceeds to the `print` statement and finally reaches `nop`, which represents the end of the execution. Since the first statement of P_1 is the assignment on Line 1, the initial state of P_1 is represented by assign_1. We transform other programs P_2, P_3, and P_4 into LCTRSs in the same manner as illustrated in Figure 3.

We assume that the `print` statement for the final result is given in advance. In addition, we consider that other `print` statements defined by students are introduced for debugging. For this reason, regarding `print` statements, we ignore printed strings when transforming them into constrained rewrite rules, while we do not ignore the `print` statements as a whole. The `print` statement for the final result should be examined. We will discuss how to ignore `print` statements inserted for debugging in the next section.

$$\mathcal{R}_2 = \left\{ \begin{array}{rll} \mathsf{assign}_1 \to \mathsf{assign}_2(3.0) & & \\ \mathsf{assign}_2(p) \to \mathsf{assign}_3(p,q) & [\, q = 2.0 *. \sqrt{3.0} \,] \\ \mathsf{assign}_3(p,q) \to \mathsf{while}_1(p,q,6) & & \\ \mathsf{while}_1(p,q,n) \to \mathsf{assign}_4(p,q,n) & [\, n < 3072 \,] \\ \mathsf{while}_1(p,q,n) \to \mathsf{nop}(p,q,n) & [\, \neg(n < 3072) \,] \\ \mathsf{assign}_4(p,q,n) \to \mathsf{assign}_5(p,q,n') & [\, n' = n * 2 \,] \\ \mathsf{assign}_5(p,q,n) \to \mathsf{assign}_6(p,q',n) & [\, q' = 2.0 *. p *. q /. (p +. q) \,] \\ \mathsf{assign}_6(p,q,n) \to \mathsf{print}_1(p',q,n) & [\, p' = \sqrt{p *. q} \,] \\ \mathsf{print}_1(p,q,n) \to \mathsf{while}_1(p,q,n) & & \end{array} \right\}$$

$$\mathcal{R}_3 = \left\{ \begin{array}{rll} \mathsf{assign}_1 \to \mathsf{assign}_2(3.0) & & \\ \mathsf{assign}_2(p) \to \mathsf{assign}_3(p,q) & [\, q = 2.0 *. \sqrt{3.0} \,] \\ \mathsf{assign}_3(p,q) \to \mathsf{forinit}_1(p,q,6) & & \\ \mathsf{forinit}_1(p,q,n) \to \mathsf{for}_1(p,q,n,0) & & \\ \mathsf{for}_1(p,q,n,i) \to \mathsf{assign}_4(p,q,n,i) & [\, i < 9 \,] \\ \mathsf{for}_1(p,q,n,i) \to \mathsf{print}_1(p,q,n) & [\, \neg(i < 9) \,] \\ \mathsf{assign}_4(p,q,n,i) \to \mathsf{assign}_5(p,q,n',i) & [\, n' = n * 2 \,] \\ \mathsf{assign}_5(p,q,n,i) \to \mathsf{assign}_6(p,q',n,i) & [\, q' = 2.0 *. p *. q /. (p +. q) \,] \\ \mathsf{assign}_6(p,q,n,i) \to \mathsf{for}_1(p',q,n,i') & [\, p' = \sqrt{p *. q} \wedge i' = i + 1 \,] \\ \mathsf{print}_1(p,q,n) \to \mathsf{nop}(p,q,n) & & \end{array} \right\}$$

$$\mathcal{R}_4 = \left\{ \begin{array}{rll} \mathsf{assign}_1 \to \mathsf{assign}_2(3.0) & & \\ \mathsf{assign}_2(p) \to \mathsf{assign}_3(p,q) & [\, q = 2.0 *. \sqrt{3.0} \,] \\ \mathsf{assign}_3(p,q) \to \mathsf{while}_1(p,q,6) & & \\ \mathsf{while}_1(p,q,n) \to \mathsf{assign}_4(p,q,n) & [\, n \leq 3072 \,] \\ \mathsf{while}_1(p,q,n) \to \mathsf{print}_1(p,q,n) & [\, \neg(n \leq 3072) \,] \\ \vdots & & \\ \mathsf{print}_1(p,q,n) \to \mathsf{nop}(p,q,n) & & \end{array} \right\}$$

Figure 3: LCTRSs $\mathcal{R}_2$, $\mathcal{R}_3$, and $\mathcal{R}_4$ for the programs P_2, P_3, and P_4, respectively

5 Comparing Rewrite Sequences

In this section, we compare rewrite sequences obtained from LCTRSs to detect differences in execution.

Conventional equivalence verification checks whether two programs produce the same output for the same input. However, this is not sufficient for programming assignments, since differences in computation processes or violations of coding requirements cannot be detected. In our approach, both the model answer (i.e., correct) program and the program submitted by the students are transformed into LCTRSs, and their rewrite sequences are compared from the same initial state. If the rewrite sequences are equivalent, the submitted program can be regarded as correct. Otherwise, the subsequences where the sequences differ can be provided as feedback.

5.1 Comparison Algorithm for Rewrite Sequences

We first show an algorithm to compare rewrite sequences of two LCTRSs, one of which is a model answer program, and the other of which is a program to be graded. Note that the rewrite sequences start

with the terms representing the initial states of the corresponding programs. We assume that the answer program includes no auxiliary `print` statement for, e.g., debugging, and the initial term is terminating and maximal rewrite sequences from the initial term are unique.

Programs to be graded may contain statements that do not affect the final result. For example, `print` statements may be used to display intermediate results for debugging. Therefore, in this paper, such irrelevant `print` statements are ignored when we compare rewrite sequences.

Since rewrite sequences can be considered lists of terms, we use a Haskell-like list notation—the empty list `[]`, the right-associative constructor operator :, and list literals $[t_1,\ldots,t_n]$—for rewrite sequences in a formal statement of the comparison algorithm. For returned values of the algorithm, we use the option type (`Maybe` type in Haskell) of term pairs: `Nothing` and `Maybe` (u_1,u_2).

The comparison algorithm shown below takes two finite rewrite sequences τ_1,τ_2 such that

- τ_i is a rewrite sequence of an LCTRS $\mathcal{R}_i$ obtained from a Python program in the way shown in Section 4,
- τ_i starts with the term representing the initial state of the corresponding Python program,
- $\mathcal{R}_1$ is obtained from a model answer program without any auxiliary `print` statement, and
- τ_1 ends with a normal form of $\mathcal{R}_1$.

The algorithm returns values with the option type of term pairs: `Nothing` or `Maybe` (u_1,u_2) with $u_1 \neq u_2$.

Definition 5.1 (comparison algorithm *Diff*) *We define a comparison algorithm Diff, which takes two finite rewrite sequences τ_1,τ_2 and returns the option type of term pairs, recursively as follows:*

$$
\begin{array}{rcl}
\mathit{Diff}([],[]) & = & \texttt{Nothing} \\
\mathit{Diff}(u_1\!:\!us_1,u_2\!:\!us_2) & = & \begin{cases}
\mathit{Diff}(us_1,us_2) & \text{if } u_1 = u_2 \\
\mathit{Diff}(u_1\!:\!us_1,us_2) & \text{if } u_1 \neq u_2 \text{ and } u_2 = \mathsf{print}_j(\ldots) \text{ for some } j>0 \\
\texttt{Maybe}\ (u_1,u_2) & \text{otherwise}
\end{cases}
\end{array}
$$

Note that either $\mathit{Diff}([],us)$ or $\mathit{Diff}(us,[])$ with $us \neq []$ is never called, and thus not defined, because any rewrite sequence corresponding to a terminating execution ends with a term rooted by `nop`. In the case with $u_1\!:\!us_1$ and $u_2\!:\!us_2$, if u_1 and u_2 are equivalent, then *Diff* examines whether us_1 and us_2 are equivalent or not; if u_1 and u_2 are different and u_2 represents a state just before a `print` statement, then u_2 is considered a location just before a `print` statement introduced for debugging, and thus *Diff* ignores u_2; otherwise, we consider the sequences $u_1\!:\!us_1$ and $u_2\!:\!us_2$ to be different because of the witness (u_1,u_2).

Regarding concrete examples shown in the rest of this section, for readability, we denote the rewrite sequence $[t_1,\ldots,t_n]$ of $\mathcal{R}$ by $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_n$ as usual. For the comparison with P_1, we consider the following rewrite sequence τ_1 of $\mathcal{R}_1$ for the program P_1, which determines the range of π for a regular 3072-gon:

$$
\begin{array}{rcll}
\tau_1 : \mathsf{assign}_1 & \to_{\mathcal{R}_1} & \mathsf{assign}_2(3.0) \\
& \to_{\mathcal{R}_1} & \mathsf{assign}_3(3.0,\ 3.464102) \\
& \to_{\mathcal{R}_1} & \mathsf{while}_1(3.0,\ 3.464102,\ 6) \\
& \to_{\mathcal{R}_1} & \mathsf{assign}_4(3.0,\ 3.464102,\ 6) \\
& \to_{\mathcal{R}_1} & \cdots \\
& \to_{\mathcal{R}_1} & \mathsf{while}_1(3.141592,\ 3.141594,\ 3072) \\
& \to_{\mathcal{R}_1} & \mathsf{print}_1(3.141592,\ 3.141594,\ 3072) \\
& \to_{\mathcal{R}_1} & \mathsf{nop}(3.141592,\ 3.141594,\ 3072)
\end{array}
$$

5.2 Differences in Rewrite Sequences for Different Constructs

First, we compare the correct program P_1 with the incorrect program P_3 with `for` loop. The rewrite sequence τ_3 of $\mathcal{R}_3$ for P_3 is as follows:

$$\begin{aligned}
\tau_3 \ : \ \mathsf{assign}_1 &\to_{\mathcal{R}_3} \mathsf{assign}_2(3.0) \\
&\to_{\mathcal{R}_3} \mathsf{assign}_3(3.0,\ 3.464102) \\
&\to_{\mathcal{R}_3} \mathsf{forinit}_1(3.0,\ 3.464102,\ 6) \\
&\to_{\mathcal{R}_3} \mathsf{for}_1(3.0,\ 3.464102,\ 6,\ 0) \\
&\to_{\mathcal{R}_3} \mathsf{assign}_4(3.0,\ 3.464102,\ 6,\ 0) \\
&\to_{\mathcal{R}_3} \cdots \\
&\to_{\mathcal{R}_3} \mathsf{for}_1(3.141592,\ 3.141594,\ 3072,\ 9) \\
&\to_{\mathcal{R}_3} \mathsf{print}_1(3.141592,\ 3.141594,\ 3072) \\
&\to_{\mathcal{R}_3} \mathsf{nop}(3.141592,\ 3.141594,\ 3072)
\end{aligned}$$

The above rewrite sequence reaches the same final term $\mathsf{nop}(3.141592,\ 3.141594,\ 3072)$ as that of P_1. This is because both programs perform the same number of iterations. However, the sequences themselves are not equivalent, since the function symbols appearing in the intermediate terms are different. In fact, we have that

$$\mathit{Diff}(\tau_1, \tau_3) = \texttt{Maybe}\ (\mathsf{while}_1(3.0, 3.464102, 6), \mathsf{forinit}_1(3.0, 3.464102, 6))$$

Therefore, P_3 cannot be considered correct w.r.t. the assignment specification.

In this case, the different subsequence corresponds to the loop part of the program, and we can provide feedback to indicate where the implementations differ: The difference between while_1 and $\mathsf{forinit}_1$ indicates that a `for` statement is used instead of a `while` statement.

5.3 Differences in Rewrite Sequences for Different Loop Conditions

Next, we compare the correct program P_1 with the incorrect program P_4 which includes the incorrect loop condition. The rewrite sequence τ_4 of $\mathcal{R}_4$ for P_4 is as follows:

$$\begin{aligned}
\mathsf{assign}_1 &\to_{\mathcal{R}_4} \mathsf{assign}_2(3.0) \\
&\to_{\mathcal{R}_4} \mathsf{assign}_3(3.0,\ 3.464102) \\
&\to_{\mathcal{R}_4} \mathsf{while}_1(3.0,\ 3.464102,\ 6) \\
&\to_{\mathcal{R}_4} \mathsf{assign}_4(3.0,\ 3.464102,\ 6) \\
&\to_{\mathcal{R}_4} \cdots \\
&\to_{\mathcal{R}_4} \mathsf{while}_1(3.141592,\ 3.141594,\ 3072) \\
&\to_{\mathcal{R}_4} \mathsf{assign}_4(3.141592,\ 3.141594,\ 3072) \\
&\to_{\mathcal{R}_4} \cdots \\
&\to_{\mathcal{R}_4} \mathsf{while}_1(3.141593,\ 3.141593,\ 6144) \\
&\to_{\mathcal{R}_4} \mathsf{print}_1(3.141593,\ 3.141593,\ 6144) \\
&\to_{\mathcal{R}_4} \mathsf{nop}(3.141593,\ 3.141593,\ 6144)
\end{aligned}$$

The program P_4 satisfies the coding requirement and computes the range of π when $N = 3072$ during execution. However, since the loop condition is different, the number of iterations is larger by one, and the lengths of the rewrite sequences are not equivalent. As a result, the final term reached by P_4 also differs from that of P_1. In fact, we have that

$$\mathit{Diff}(\tau_1, \tau_4) = \texttt{Maybe}\ (\mathsf{nop}(3.141592, 3.141594, 3072), \mathsf{while}_1(3.141593, 3.141593, 6144))$$

Therefore, P_4 cannot be considered correct w.r.t. the assignment specification.

In this case, the different subsequence corresponds to the part of the program after the additional loop iterations, and we can provide feedback to indicate where the implementations differ: The difference in the length indicates that the loop condition is not semantically equivalent to the expected one.

5.4 Differences in Rewrite Sequences among Correct Programs

Finally, we compare the correct program P_1 with another correct program P_2. The rewrite sequence τ_2 of $\mathcal{R}_2$ for P_2 is as follows:

$$\begin{aligned}
\tau_2 \,:\, \mathsf{assign}_1 &\to_{\mathcal{R}_2} \mathsf{assign}_2(3.0)\\
&\to_{\mathcal{R}_2} \mathsf{assign}_3(3.0,\ 3.464102)\\
&\to_{\mathcal{R}_2} \mathsf{while}_1(3.0,\ 3.464102,\ 6)\\
&\to_{\mathcal{R}_2} \mathsf{assign}_4(3.0,\ 3.464102,\ 6)\\
&\to_{\mathcal{R}_2} \cdots\\
&\to_{\mathcal{R}_2} \mathsf{print}_1(3.105829,\ 3.215390,\ 12)\\
&\to_{\mathcal{R}_2} \cdots\\
&\to_{\mathcal{R}_2} \mathsf{print}_1(3.141592,\ 3.141594,\ 3072)\\
&\to_{\mathcal{R}_2} \mathsf{while}_1(3.141592,\ 3.141594,\ 3072)\\
&\to_{\mathcal{R}_2} \mathsf{nop}(3.141592,\ 3.141594,\ 3072)
\end{aligned}$$

In this case, ignoring all the intermediate `print` statements, the rewrite sequence τ_2 coincides with τ_1 in terms of length, the final term, and all intermediate terms up to the point just before the final `print` and the termination of the loop. However, the order of the final `print` and the loop exit is reversed, and therefore the rewrite sequences do not coincide as a whole. In fact, we have that

$$\mathit{Diff}(\tau_1,\tau_2) = \texttt{Maybe}\ (\mathsf{print}_1(3.141592,3.141594,3072),\mathsf{while}_1(3.141593,3.141593,6144))$$

The above example shows that, if correctness is determined only by *Diff*, even correct programs may be judged as incorrect w.r.t. the assignment specification. A solution is to swap the final `print` statement and the successor element: $\mathit{Diff}(u_1:us_1,u_2:us_2) = \mathit{Diff}(u_1:us_1,u_3:u_2:us_3)$ if $u_1 \neq u_2\ u_2 = \mathsf{print}_j(\ldots)$ for some $j > 0$, and there is no term of the form $\mathsf{print}_k(\cdots)$ in us_2, where $us_2 = u_3:us_3$. For this modification, we have that $\mathit{Diff}(\tau_1,\tau_2) = \texttt{Nothing}$. On the other hand, instead of relying on a single correct sequence, it may be necessary to prepare multiple correct sequences.

6 Conclusion

In this paper, we proposed a method to support grading of introductory programming assignments based on the comparison of rewrite sequences obtained from the LCTRSs transformed from Python programs. We compared rewrite sequences of the model answer program and submitted programs. By this comparison, we captured differences not only in final outputs but also in computation processes, such as differences in constructs, loop conditions, and execution order.

As future work, we will extend the method to detect violations of assignment specifications more precisely, such as cases where required variables are not used in loop conditions. In addition, we will implement an interpreter of LCTRSs with the theories of integers and floating point numbers.

Acknowledgment We thank the anonymous reviewers for their valuable feedback, which improved the paper.

References

[1] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1145/505863.505888.

[2] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2016): *The Satisfiability Modulo Theories Library (SMT-LIB)*. `https://www.SMT-LIB.org`.

[3] Martin Brain, Cesare Tinelli, Philipp Rümmer & Thomas Wahl (2015): *An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic*. In: *Proceedings of the 22nd IEEE Symposium on Computer Arithmetic*, IEEE, pp. 160–167, doi:10.1109/ARITH.2015.26. Available at `https://doi.org/10.1109/ARITH.2015.26`.

[4] Ştefan Ciobâcă, Dorel Lucanu & Andrei-Sebastian Buruiană (2023): *Operationally-based program equivalence proofs using LCTRSs*. *Journal of Logical and Algebraic Methods in Programming* 135, pp. 1–22, doi:10.1016/j.jlamp.2023.100894.

[5] Ştefan Ciobâcă & Dorel Lucanu (2018): *A Coinductive Approach to Proving Reachability Properties in Logically Constrained Term Rewriting Systems*. In Didier Galmiche, Stephan Schulz & Roberto Sebastiani, editors: *Proceedings of the 9th International Joint Conference on Automated Reasoning*, *Lecture Notes in Computer Science* 10900, Springer, pp. 295–311, doi:10.1007/978-3-319-94205-6_20.

[6] Carsten Fuhs, Cynthia Kop & Naoki Nishida (2017): *Verifying Procedural Programs via Constrained Rewriting Induction*. *ACM Transactions on Computational Logic* 18(2), pp. 14:1–14:50, doi:10.1145/3060143.

[7] Yoshiaki Kanazawa & Naoki Nishida (2019): *On Transforming Functions Accessing Global Variables into Logically Constrained Term Rewriting Systems*. In Joachim Niehren & David Sabel, editors: *Proceedings of the 5th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, *Electronic Proceedings in Theoretical Computer Science* 289, Open Publishing Association, pp. 34–52.

[8] Misaki Kojima, Naoki Nishida & Yutaka Matsubara (2025): *Transforming concurrent programs with semaphores into logically constrained term rewrite systems*. *Journal of Logical and Algebraic Methods in Programming* 143, pp. 1–23, doi:10.1016/j.jlamp.2024.101033.

[9] Cynthia Kop & Naoki Nishida (2013): *Term Rewriting with Logical Constraints*. In Pascal Fontaine, Christophe Ringeissen & Renate A. Schmidt, editors: *Proceedings of the 9th International Symposium on Frontiers of Combining Systems*, *Lecture Notes in Computer Science* 8152, Springer, pp. 343–358, doi:10.1007/978-3-642-40885-4_24.

[10] Cynthia Kop & Naoki Nishida (2015): *Constrained Term Rewriting tooL*. In Martin Davis, Ansgar Fehnker, Annabelle McIver & Andrei Voronkov, editors: *Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, *Lecture Notes in Computer Science* 9450, Springer, pp. 549–557, doi:10.1007/978-3-662-48899-7_38.

[11] Ayuka Matsumi, Naoki Nishida, Misaki Kojima & Donghoon Shin (2023): *On Singleton Self-Loop Removal for Termination of LCTRSs with Bit-Vector Arithmetic*. In Akihisa Yamada, editor: *Proceedings of the 19th International Workshop on Termination*, pp. 1–6, doi:10.48550/arXiv.2307.14094.

[12] Dragana Milovančević, Carsten Fuhs & Viktor Kunčak (2025): *Proving Termination of Scala Programs by Constrained Term Rewriting*. In Cynthia Kop & Janis Voigtländer, editors: *Informal Proceedings of the 11th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, pp. 13–25. Available at `https://wpte2025.github.io/pre-proceedings.pdf#page=15`.

[13] Naoki Nishida & Sarah Winkler (2018): *Loop Detection by Logically Constrained Term Rewriting*. In Ruzica Piskac & Philipp Rümmer, editors: *Proceedings of the 10th Working Conference on Verified Software: Theories, Tools, and Experiments*, *Lecture Notes in Computer Science* 11294, Springer, pp. 309–321, doi:10.1007/978-3-030-03592-1_18.

[14] Enno Ohlebusch (2002): *Advanced Topics in Term Rewriting*. Springer, doi:10.1007/978-1-4757-3661-8.

[15] Uday S. Reddy (1990): *Term Rewriting Induction*. In Mark E. Stickel, editor: *Proceedings of the 10th International Conference on Automated Deduction*, *Lecture Notes in Computer Science* 449, Springer, pp. 162–177, doi:10.1007/3-540-52885-7_86.

[16] Philipp Rümmer & Thomas Wahl (2010): *An SMT-LIB Theory of Binary Floating-Point Arithmetic*. In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, pp. 1–14. Available at `http://www.philipp.ruemmer.org/publications/smt-fpa.pdf`.

Tactic-driven code fusion

Katarzyna Marek
EPFL, Switzerland
katarzyna.marek@epfl.ch

Clément Pit-Claudel
EPFL, Switzerland
clement.pit-claudel@epfl.ch

Code fusion, also called deforestation, is a compiler-optimization technique that removes intermediate data structures. Though it has been studied in the literature for over 30 years, it is rarely used in production compilers due to the many challenges that automatic deforestation faces. We believe that many of these issues can be solved if we think about deforestation as a guided process instead. In this work-in-progress report, we describe the user-controlled system we envision and give preliminary results from working on it. More specifically, we present a small, domain-specific language for deforestation and give intuition for the way many of previous deforestation techniques can be expressed in it.

1 Introduction

Functional programs make heavy use of intermediate data structures. For example, most phases of a compiler produce an intermediate AST only so that it can be consumed by the next phase. There is a trade-off here: splitting the compilation into many small phases gives us better modularity and readability, but it increases the number of AST traversals and the amount of memory allocated. This problem is well known, and it is not specific to compilers: all functional programs face this trade-off. Many solutions have been proposed, with varying degrees of generality and applicability [37, 12, 19, 26]. One of them is deforestation, also called fusion.

Deforestation [37] is a compiler-optimization technique that aims to remove redundant intermediate tree-like data structures by fusing neighboring operations. When done well, this results in a smaller memory footprint, lower cache pressure, and fewer traversals. For example, consider the following functions:

```
def sum := l => l is
  of [] => 0
  of x::xs => x + (sum xs)
// sum [1, 3, 6] ⟶* 10
```

```
def range := (m, n) =>
  if m > n then []
  else m::(range (m + 1) n)
// range 2 4 ⟶* [2, 3, 4]
```

We present examples in our custom, untyped language $\mathcal{L}$ that can be thought of as lambda calculus with additional syntax for ADTs and primitive integers. If we combine them to sum numbers from 1 to n (`sum (range 1 n)`), then `range` creates a temporary list that is immediately consumed by `sum`. This code can be deforested into a more efficient form that does not use lists at all:

```
let h = (m, n) => if m > n then 0 else m + (h (m + 1) n) in h 1 n
```

As with most program transformations, deforestation (and related techniques) bring uneven benefits to different programs or program fragments. Additionally, variants that we have looked at suffer from some of the following issues:

- Advanced algorithms often do not scale to large codebases [14, 4].
- The impact of complex transformations can be hard to predict: they suffer from optimization cliffs, where even small source changes can vastly impact the result.

- More predictable techniques can give limited benefits or miss optimizations [12, 33].
- It is not always clear what *should* be deforested. Uncontrolled deforestation can lead to code-size increases, and even to duplicated computations [28, 4].

Consequently, the literature reports mixed results for advanced deforestation techniques [4], and most production compilers do not implement a general deforestation optimization. Instead, users resort to ad-hoc solutions such as using special visitor patterns or custom data structures and libraries [18, 19].
To tackle these problems, we believe that deforestation should be thought of as a *guided* process instead. We further make the following three claims:

- Deforestation should be user-driven and fine-grained: users should be able to decide what should be deforested and control the choice of techniques.
- Many existing deforestation algorithms are special cases that could be expressed as strategies (meta-programs) within a small domain specific language for deforestation.
- To address optimization cliffs, users should be allowed to state deforestation objectives, and compilation could fail if these objectives are not met.

User-control As we already underlined, the benefits of deforestation are uneven for different code fragments, and not all deforestation algorithms can handle all desired tasks. Even our initial small example, can be handled by short-cut fusion [12], a deforestation used in GHC compiler, if `sum` and `range` are implemented in a very specific way. All this brings us to the conclusion that the users need control to choose where the deforestation is applied, to target the places where it can bring the most benefits. They should also be able to choose the algorithm that best fits the task, or even implement custom deforestation strategies tailored to their specific code and requirements.

Meta-programs To illustrate the use of deforestation rules, let us look at an example: supercompiling [36] a very simple program, `orElse (head xs) v`, given the following definitions:

```
def head := l => l is                  def orElse := (x, v) => x is
  of [] => None                          of None => v
  of x::xs => Some(x)                    of Some(x) => x
// head [2, 3, 4] ⟶* Some(2)         // orElse None 3 ⟶* 3
```

A supercompiler would split the code into two cases depending on the patterns for `xs` in `head` (left), then reduce both cases and join them back into the deforested result (right):

```
case 1: orElse (head []) v ⟶* v              xs is of [] => v
case 2: orElse (head (x::xs)) v ⟶* x               of x::xs => x
```

It turns out that we can express the same transformation with a sequence of small transformations. First, we unfold the definition of `head` and beta-reduce to get

```
orElse (xs is of [] => None of x::xs => Some(x)) v
```

Next, we push the term surrounding our inlined definition of `head` under the pattern-matching branches

```
xs is of [] => orElse None v of x::xs => orElse Some(x) v
```

Finally, we simplify the calls to `orElse` using unfolding and more reduction rules and we get the same result: `xs is of [] => v of x::ys => x`.

Deforestation objectives There is a very clear deforestation objective for our initial example `sum ( range 1 n)`: there should be no lists in that scope. Current compilers give no explicit way for the user to ensure that this objective is met. In most cases, the only options are to use (advisory) inlining annotations and hope that the compiler optimizes properly, or to rewrite the code manually.

Contributions The aim of this work-in-progress report is to start supporting these three claims. So far, we have only focused on the first two and in this paper we present evidence for the first one and partial evidence for the second. More specifically:

- We present a small language for deforestation that can be used to create targeted deforestation scripts.
- We give some intuition of how some previous deforestation results can be reproduced in this small language for deforestation.
- We report on a proof-of-concept implementation of this deforestation language.

The rest of this paper is organized as follows. In Section 2, we explain our approach and present our deforestation language formally using the running example of `orElse (head xs) v`. In Section 3, we present a larger deforestation example taken from Hamilton's distillation paper [14]. Finally, in Section 4, we discuss the current limitations and plans for further work, and we briefly summarize how our implementation of the system compares to the theory presented in this paper. We conclude with related work in Section 5.

2 Tactic-driven deforestation

Rather than designing complex deforestation algorithms in isolation, we aim to create a language of deforestation rules we call *tactics*, that can be composed into more sophisticated tactics. To distinguish these composite tactics from primitive ones, we sometimes refer to them as *meta-programs*. We use terminology from the world of interactive theorem provers, where individual transformations and multi-step transformations are both called "tactics" (in contrast, individual transformations are often called "rules" in the rewriting litterature, and "strategies" are the programs that combine them). To this aim, we introduce a simple, untyped, pure, call-by-name functional language that we call the $\mathcal{L}$ language, and a separate language of tactics to manipulate $\mathcal{L}$ terms.

2.1 Syntax and semantics of the $\mathcal{L}$ language

Figures 1a and 1b present the syntax and the small step call-by-name semantics of the language. Note that we use values (denoted by v) to achieve determinism of the reduction rules. We use standard notations `(`x_1`,...,`x_n`) => `e for `(`x_1`) =>...(`x_n`) => `e, `[]` for `Nil()`, x`::`xs for `Cons(`x`, `xs`)`, `[`x_1`,...,`x_n`]` for `Cons(`x_1`,...Cons(`x_n`, Nil())...)`, and **`if`** e **`then`** e_1 **`else`** e_2 for e **`is of`** `True() =>` e_1 **`of`** `False() =>` e_2 .

2.2 Interactive deforestation

In this section we show in detail how to deforest our introductory example, `(xs, v) => orElse ( head xs) v`, while going through the important parts of our approach.

Variable	x			
Integer constant	n			
Integer operation	$op_{\mathbb{Z}}$	::=	`*` | `+` | `-` | `<`	
Binary operation	op	::=	$op_{\mathbb{Z}}$ | `=?`	
Pattern-match branch	br	::=	`of` c`(`x_1`,` x_2`, ...)` `=>` e	
Expression	e	::=	| x	
			| n	
			| e op e	
			| c`(`e_1`,` e_2`, ...)`	constructor application
			| `let` x `=` e_1 `in` e_2	let binding
			| x `=>` e	abstraction
			| e_1 e_2	application
			| `fix(`f `=>` e`)`	fixpoint
			| e `is` br_1 br_2 `...`	pattern match

(a) $\mathcal{L}$ language used in the project.

$$\frac{f \texttt{ := } e \in \Gamma}{f \longrightarrow e}\ \text{Def} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ op\ e_2 \longrightarrow e_1'\ op\ e_2}\ \text{OpL} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ op\ e_2 \longrightarrow v_1\ op\ e_2'}\ \text{OpR} \qquad \frac{[\![op_{\mathbb{Z}}]\!]\ n_1\ n_2 = l}{n_1\ op_{\mathbb{Z}}\ n_2 \longrightarrow l}\ \text{OpInt}$$

$$\frac{e \longrightarrow e'}{c(v_1,\ldots,v_n,\ e,e_1,\ldots,e_m) \longrightarrow c(v_1,\ldots,v_n,\ e',e_1,\ldots,e_m)}\ \text{RedArg}$$

$$\frac{v_1 = v_2}{v_1\ \texttt{=?}\ v_2 \longrightarrow \texttt{True()}}\ \text{OpEq} \qquad \frac{v_1 \neq v_2}{v_1\ \texttt{=?}\ v_2 \longrightarrow \texttt{False()}}\ \text{OpNotEq} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{AppL}$$

$$\frac{}{\texttt{((}x\texttt{)}\ \texttt{=>}\ e_1\texttt{)}\ e_2 \longrightarrow e_1[x := e_2]}\ \text{Beta} \qquad \frac{e \longrightarrow e'}{e\ \texttt{is}\ br_i\ldots \longrightarrow e'\ \texttt{is}\ br_i\ldots}\ \text{CaseL}$$

$$\frac{}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \longrightarrow e_2[x := e_1]}\ \text{LetReduce} \qquad \frac{}{\texttt{fix(}f\ \texttt{=>}\ e_1\texttt{)} \longrightarrow e_1[f := \texttt{fix(}f\ \texttt{=>}\ e_1\texttt{)}]}\ \text{Fix}$$

$$\frac{}{c(e_1,\ldots,e_n)\ \texttt{is of}\ c(x_1,\ldots,x_n)\ \texttt{=>}\ e\ \ br_2\ldots br_k \longrightarrow e[x_1 := e_1,\ldots,x_n := e_n]}\ \text{CaseMatch}$$

$$\frac{c' \neq c\ \texttt{ or }\ k \neq n}{c'(v_1,\ldots,v_k)\ \texttt{is of}\ c(x_1,\ldots,x_n)\ \texttt{=>}\ e\ \ br_2\ldots br_k \longrightarrow c'(v_1,\ldots,v_k)\ \texttt{is}\ br_2\ldots br_k}\ \text{CaseNoMatch}$$

(b) Small-step semantics for $\mathcal{L}$. We use v to denote values (terms that cannot be further reduced). The global context Γ is constant and left implicit in the reduction rules. By $[\![op_{\mathbb{Z}}]\!]$ we understand the usual interpretation of $op_{\mathbb{Z}}$ on integers and by = equality on integers and data structures. We use constructors `True` and `False` to denote logical true and false respectively, thus $[\![<]\!]$ `1 2` is `True()`. Finally, in the substitution $e[x_1 := e_1,\ldots,x_n := e_n]$ we assume that required α-renaming is performed on e, so none of the free variables of terms $x_1,\ldots,x_n$ become bound in $e[x_1 := e_1,\ldots,x_n := e_n]$.

Deforestation in our system is an interactive process of transforming a term using a predefined set of tactics. Each one operates on a *goal*, which pairs the *subject* term e being rewritten, and a context. The context is a four-element tuple (Γ, Θ, Σ, e_0):

- Γ is the global context of function definitions: a collection of name-to-expression bindings. In our example, this will be {`def head := ...`, `def orElse := ...`}.
- Θ is the global context of equalities on terms: a collection of equalities between terms that are closed under Γ (all their free variables are contained in Γ). Since equality is symmetric, if we have $e \equiv e' \in \Theta$ then $e' \equiv e$ is also implicitly included in Θ. In our example Θ is empty.
- Σ is the local context of function and fixpoint definitions and any local variables. Initially, this context is empty.
- e_0 is the initial shape of the term e. Intuitively, if we create an equality during deforestation, e_0 will be the left-hand side of the equality and the current subject will be the right-hand side. For our example, e_0 will be `(xs, v) => orElse (head xs) v`.

The initial subject equals e_0 and should be closed under Γ.

2.2.1 Preconditions

Path freshness We assume and maintain the property of path freshness for terms and contexts. A context is fresh if all the names in the context are unique. A term is path-fresh with respect to the context $C = (\Gamma, \Theta, \Sigma)$ if bound names do not reuse any of the free variables' names and on each path in the expression tree there are no two bindings with the same name. Intuitively, this means that no variable name can be shadowed. For example, if we naively unfold the definition of `head` and beta-reduce in our running example, we get

```
(xs, v) => orElse (xs is of [] => None of x::xs => Some(x)) v
```

which is not path-fresh due to the repeated use of `xs` on the same path. By $\mathit{freshen}(t, C)$ we denote the term t alpha-renamed so it is path-fresh with respect to C. We also omit the context and write $\mathit{freshen}(t)$ if it is implicitly understood which context the freshness property refers to.

Subterms and local contexts To allow for application of deforestation tactics on subterms we introduce some notions that will allow us to talk about them. We denote by $t|_p$ a subterm of t at *path* p where a path is a sequence of numbers that starting from the root node select a child in t's AST (called *position* in [2]). We use Λ to denote an empty sequence, which corresponds to the root term. For example, for t equal `orElse (head xs) v` the subterm at position 1.1, denoted by $t|_{1.1}$, is `xs`. By $t[t']_p$ we denote a substitution of subterm at path p with t'.

Additionally, we define *definition contexts*: *def_ctx*`(p, t)` as the sequence of all variables and definitions in scope for the subterm at position p, e.g., for t defined as

```
let x := 3
in fix(f => (xs) => xs is
    of [] => x + 3 + 4
    of y::ys => let z := y + 3 in z + f(ys))
```

def_ctx`(2.1.1.2, t)` is `(def x := 3, fix f := fix(f => ...), val xs)` (where $t|_{2.1.1.2}$ is `x + 3 + 4`). We distinguish three kinds of entries: bindings defined using `let` (preceded by the `def` keyword), fixpoints with a recursive definition (preceded by the `fix` keyword), and variables without definitions (preceded by the `val` keyword). Distinguishing `fix` and `def` is important for soundness: we disallow re-folding fixpoints to avoid accidental creation of non-well-founded recursion (Section 2.2.2).

Recursion Though for simplicity our running example does not have any recursion, the $\mathcal{L}$ language does support it. Recursion is modeled using the `fix` operator, which conceptually ensures that the recursion is well-founded. We do not check this in our prototype implementation and instead leave it to the user to warrant the well-foundedness of fixpoints. Well-foundedness is needed to ensure that `f = fix(F)` is uniquely defined (this lets us rely on fixpoint equality which substitutes traditional induction).

We do not have a notation for mutual recursion yet: function definitions in the global context are ordered, and each definition can only refer to previously defined symbols. Additionally, to avoid direct recursion, the variable x in **let** x := e_1 **in** e_2 is not in scope for e_1.

Since the well-foundedness of `fix` is crucial for correctness of our tactics, we include it as an explicit (unchecked) side-condition in the definition of the tactics (see Section 2.2.2) by writing $wf(f \Rightarrow e)$.

2.2.2 Deforestation tactics

To illustrate our rudimentary tactics, let us deforest our running example following the sketch provided in the introduction. We give a more realistic case study in Section 3. We start with the following subject:

```
(xs, v) => orElse (␣head␣ xs) v
```

First, we want to reveal the definition of `head`. We reach for the Unfold tactic, which replaces a function with its definition. However, because our tactics work on the whole subject, we must first use the Focus tactic: Focus lets us perform a sequence of tactics (excluding PushGoal) on a chosen subterm of the subject, and then embed the result back into the original context (for ease of reading, we indicate the target of the next transformation using ␣brackets␣, and changed code in `teal`). Combining Focus and Unfold (left), we obtain the following new subject (right):

$$\frac{(f \texttt{ := } e) \in \Gamma \cup \Sigma}{\begin{array}{l}(\Theta, \Sigma, e_0) \vdash f \rightsquigarrow \\ (\Theta, \Sigma, e_0) \vdash \mathit{freshen}(e)\end{array}}\ \textsc{Unfold}$$

```
(xs,v) =>
  orElse ␣((l is
    of [] => None
    of x::xs1 => Some(x)) xs)␣ v
```

Note that the inner `xs` was renamed to `xs1` as a result of *freshen*. Next, we would like to beta-reduce the newly-unfolded definition. Again, we can use Focus to concentrate on the correct subterm and use the Reduce tactic (left) to get the result on the right

$$\frac{\emptyset \vdash e \longrightarrow e'}{\begin{array}{l}(\Theta, \Sigma, e_0) \vdash e \rightsquigarrow \\ (\Theta, \Sigma, e_0) \vdash \mathit{freshen}(e')\end{array}}\ \textsc{Reduce}$$

```
(xs,v) =>
  ␣orElse (xs is
    of [] => None
    of x::xs1 => Some(x)) v␣
```

Now we want to push the surroundings under the pattern match (a form of commuting conversion). To do that we need our subject to have a concrete shape – a function applied to a pattern match, which allows us to push the function application into branches. To get that shape we use the Abstract tactic (left), so the pattern-match becomes an argument to a function that embeds it back in the context (right):

$$\frac{\forall x \in fv(e').x \in (\Theta \cup \Sigma)}{\begin{array}{l}(\Theta, \Sigma, e_0) \vdash e \rightsquigarrow \\ (\Theta, \Sigma, e_0) \vdash \texttt{((}x\texttt{) => }e[e' := x]\texttt{) }e'\end{array}}\ \textsc{Abstract}$$

```
(xs,v) => ␣((m => orElse m v)(
  xs is
    of [] => None
    of x::xs1 => Some(x))␣
```

Note that the precondition to Abstract ensures that we cannot create free variables in the argument; for example, it stops us from incorrectly transforming our program into

```
(m => (xs,v) => orElse m v)(xs is of [] => None of x::xs1 => Some(x))
```

Now we can use the Match tactic (left) to push the function application into the branches (right):

$$\frac{}{\begin{array}{l}(\Theta,\Sigma,e_0)\vdash f(e\ \texttt{is...of}\ p_i\ \texttt{=>}\ e_i...)\rightsquigarrow\\ (\Theta,\Sigma,e_0)\vdash e\ \texttt{is...of}\ p_i\ \texttt{=>}\ \mathit{freshen}(f)e_i...\end{array}}\ \textsc{Match}$$

```
(xs,v) => xs is
  of [] =>
    (m => orElse m v) None
  of x::xs1 =>
    (m => orElse m v) Some(x)
```

Finally, by repeated use of the Focus tactic with an Unfold or Reduce as before, we can simplify the terms in both branches and get the final code: `(xs, v) => xs is of [] => v of x::xs1 => x`. We give all remaining tactics in Figure 2.

$$\frac{(\texttt{def}\ f\ \texttt{:=}\ e)\in\Gamma\cup\Sigma\qquad e\simeq e'}{(\Theta,\Sigma,e_0)\vdash e'\rightsquigarrow(\Theta,\Sigma,e_0)\vdash f}\ \textsc{Fold}$$

$$\frac{(\texttt{def}\ f\ \texttt{:=}\ \texttt{fix(}g\ \texttt{=>}\ e\texttt{))}\in\Gamma\cup\Sigma\qquad e\simeq e'}{(\Theta,\Sigma,e_0)\vdash e'[g:=f]\rightsquigarrow(\Theta,\Sigma,e_0)\vdash f}\ \textsc{FoldFix}$$

$$\frac{(e'\equiv e)\in\Theta\qquad e'\simeq e''}{\begin{array}{l}(\Theta,\Sigma,e_0)\vdash e''\rightsquigarrow\\ (\Theta,\Sigma,e_0)\vdash \mathit{freshen}(e)\end{array}}\ \textsc{Rewrite}$$

$$\frac{(f\ \texttt{:=}\ \texttt{fix(}g\ \texttt{=>}\ e\texttt{))}\in\Gamma\cup\Sigma\qquad e\simeq e'}{(\Theta,\Sigma,e_0)\vdash f\rightsquigarrow(\Theta,\Sigma,e_0)\vdash e'[g:=f]}\ \textsc{UnfoldFix}$$

$$\frac{f\notin\Gamma\cup\Sigma}{\begin{array}{l}(\Theta,\Sigma,e_0)\vdash e\rightsquigarrow\\ (\Theta,\Sigma,e_0)\vdash \texttt{let}\ f\ \texttt{:=}\ e\ \texttt{in}\ f\end{array}}\ \textsc{Extract}$$

$$\frac{}{\begin{array}{l}(\Theta,\Sigma,e_0)\vdash f(\texttt{let}\ x\ \texttt{:=}\ e_1\ \texttt{in}\ e_2)\rightsquigarrow\\ (\Theta,\Sigma,e_0)\vdash \texttt{let}\ x\ \texttt{:=}\ e_1\ \texttt{in}\ \mathit{freshen}(f\ e_2)\end{array}}\ \textsc{LetIn}$$

$$\frac{e_0'\simeq e_0\qquad \mathit{wf}(f\ \texttt{=>}\ e'\texttt{[}e_0'\ \texttt{:=}\ f\texttt{])}}{\begin{array}{l}(\Theta,\Sigma,e_0)\vdash e'\rightsquigarrow\\ (\Theta,\Sigma,e_0)\vdash \texttt{fix(}f\ \texttt{=>}\ e'\texttt{[}e_0'\ \texttt{:=}\ f\texttt{])}\end{array}}\ \textsc{Fix}$$

$$\frac{\begin{array}{l}(\Theta,\Sigma\cup \mathit{def\_ctx}(p,\ e_1),e_1|_p)\vdash e_1|_p\rightsquigarrow^*\\ (\Theta,\Sigma\cup \mathit{def\_ctx}(p,\ e_1),e_2'')\vdash e_2'\end{array}}{(\Theta,\Sigma,e_0)\vdash e_1\rightsquigarrow(\Theta,\Sigma,e_0)\vdash e_1[e_2']_p}\ \textsc{Focus}$$

$$\frac{}{\begin{array}{l}(\Theta,\Sigma,e_0)\vdash e\ \texttt{is}\ br_1...\texttt{of}\ p_k\ \texttt{=>}\ e_k...br_n\rightsquigarrow\\ (\Theta,\Sigma,e_0)\vdash e\ \texttt{is}\ br_1...\texttt{of}\ p_k\ \texttt{=>}\ e_k[e:=p_k]^{OF}...br_n\end{array}}\ \textsc{PMSubs}$$

$$\frac{}{\begin{array}{l}(\Theta,\Sigma,e_0)\vdash e_1\ \texttt{=?}\ e_2\ \texttt{of True() =>}\ e_t...br_n\rightsquigarrow\\ (\Theta,\Sigma,e_0)\vdash e_1\ \texttt{=?}\ e_2\ \texttt{of True() =>}\ e_t[e_1:=e_2]^{OF}...br_n\end{array}}\ \textsc{PMSubsEq}$$

$$\frac{}{\begin{array}{l}(\Theta,\emptyset,e_0)\vdash e\rightsquigarrow\\ (\Theta+(e_0\equiv e),\emptyset,e_0)\vdash e\end{array}}\ \textsc{CreateEq}$$

$$\frac{}{\begin{array}{l}(\Theta,\Sigma,e_0)\vdash e\rightsquigarrow\\ (\Theta,\Sigma,e)\vdash e\end{array}}\ \textsc{SetLhs}$$

$$\frac{\mathit{wf}(e')}{\begin{array}{l}(\Theta,\Sigma,e_0)\vdash e\rightsquigarrow\\ (\Theta,\emptyset,e')\vdash e'\end{array}}\ \textsc{PushGoal}$$

Figure 2: The remaining deforestation tactics. We omit the immutable variable Γ from the context. The Fold and FoldFix tactics are the opposite of Unfold ($\simeq$ denotes alpha-equivalence). Rewrite and CreateEq allow for using and creating global equalities respectively. Extract is similar to Abstract and LetIn to Match, but both work on `let` expressions. PMSubs and PMSubsEq allow for value propagation similar as in positive supercompilation [34]. PMSubs allows for rewriting the scrutinee with the pattern (by $[x:=e]^{OF}$ we mean that substitutions are optional and can be flipped), and PMSubsEq is analogous but for equality checks. SetLhs sets the initial subject to be the current one, changing the left-hand side for CreateEq, and PushGoal creates a new goal with a chosen subject resetting the local context.

3 Case study: Quadratic reverse

Most deforestation techniques lead to performance improvements that are at most linear [15, 34], but not all. Some, such as distillation [14], allow for changes to the complexity of the code: Hamilton [14] demonstrates that distillation can transform a reverse function written without an accumulator into an implementation *with* an accumulator, changing the complexity from quadratic to linear. In this section, we show how to reproduce this example in our system. (We hope to able to translate all of the distillation examples in a similar way, but we do not provide a proof of this claim yet: only a worked example, to give a sense of our system.) We start from the following definitions:

```
def rev := fix(r => l =>
  l is of [] => []
       of x::xs => (r xs) ++ [x])
```

```
def ++ := fix(a => (l, ys) =>
  l is of [] => ys
       of x::xs => x::(a xs ys))
```

We assume that equalities `(xs, ys, zs) => (xs ++ ys) ++ zs ≡ (xs, ys, zs) => xs ++ (ys ++ zs)`[1] (`assoc`) and `(xs) => xs ≡ (xs) => xs ++ []` (`addNilR`) are already in the context. We also omit calls to the Focus tactic when it is used for a single transformation on a subterm of the subject. On the left are the tactics we apply; on the right the subjects after each step.

[1] Where `x ++ y` is infix notation for `++ x y`.

0. We begin deforestation with the definition of `rev`. In Γ we have the definition of ++, in Θ the equalities `assoc` and `addNilR`, Σ is empty, and e_0 is the current subject `rev`.

```
0. (xs) => xs is
     of [] => []
     of y::ys => ⌞(rev ys) ++ [y]⌟
```

1. First, we unfold the definition of `rev` using UnfoldFix. Current subject is visible on the right.

```
1. (xs) => ⌞xs is
     of [] => []
     of y::ys =>
       ((rev ys) ++ [y]) ++ []⌟
```

2. Next, we use the Rewrite tactic with `addNilR`.

```
2. (xs) => ⌞((zs, acc) => zs is
     of [] => acc
     of y::ys =>
       ((rev ys) ++ [y]) ++ acc
   )⌟ xs []
```

3. Then we abstract the body of the fixpoint over [] and xs, using Abstract twice. This gives us a more general term in the body.

```
3. (zs, acc) => zs is
     of [] => acc
     of y::ys =>
       ((⌞rev⌟ ys) ++ [y]) ++ acc
```

4. Next, we use the Focus tactic to make a sequence of transformations (next 5 steps) on our generalized term. Note that the Focus tactic affected the context. Now we have `fix r := fix(r => ...)` and `val xs` moved into our local definition context and e_0 is equal to the current subject.

```
4.1. (zs,acc) => zs is
       of [] => acc
       of y::ys => ⌞((ys is
           of [] => []
           of u::us => (rev us) ++ [u]
         ) ++ [y]) ++ acc⌟
```

4.1 Then, we inline the definition of `rev` using the UnfoldFix tactic and beta-reduce the newly-unfolded definition.

```
4.2. (zs,acc) => zs is
       of [] => acc
       of y::ys => ys is
         of [] => ([] ++ [y]) ++ acc
         of u::us =>
```

4.2 Next, we push the `++ [y] ++ acc` into the branches using the same trick as in the previous section (ABSTRACT, MATCH, REDUCE).

4.3 Then, we rewrite with `assoc`, UNFOLD 3 times definition of ++, and reduce.

4.4. We now abstract the body of the second branch of the pattern match over `ys` and `y::acc`.

4.5. Notice that the body of the newly created abstraction is alpha-equivalent to the term from step 4, thus also to e_0. We can use that fact to create a new fixpoint definition using the FIX tactic where the newly created abstraction will become the recursive call.

5. Finally, we finish rewriting the subterm and go back to the previous goal with the subterm substituted. The deforested result is the canonical, linear implementation of a reverse function.

```
          ⌞(((rev us) ++ [u]
      ) ++ [y]) ++ acc⌟
```

```
4.3. (zs,acc) => zs is
       of [] => acc
       of y::ys => ⌞ys is
         of [] => y::acc
         of u::us =>
           ((rev us) ++ [u]) ++ y::acc⌟
```

```
4.4. ⌞(zs,acc) => zs is
       of [] => acc
       of y::ys =>
         ((vs, acc2) => vs is
           of [] => acc2
           of u::us =>
             (rev us) ++ [u]) ++ acc2
         )(ys, y::acc)⌟
```

```
4.5. fix(f => (zs,acc) => zs is
       of [] => acc
       of y::ys => f ys y::acc)
```

```
 5. (xs) => fix(f => (zs,acc) =>
       zs is of [] => acc
             of y::ys => f ys y::acc
    ) xs []
```

4 Discussion

The material we present is very much work in progress. So far we have only focused on building the set of primitive tactics for deforestation (we are currently in the process of formalizing them in the Rocq proof assistant), and we have not developed the language support (control structures, combinators) needed to express previous algorithms as strategies in our language. We are also still working on overcoming some of the limitations of our system.

Both-sides rewrites We usually think about deforestation as a directed process, moving from a term with intermediate data structures into one with fewer. Deforestation in our system works in that way most of the time, but our tactics can also be used as a rudimentary proof assistant to derive undirected equalities, by transforming both sides and meeting in the middle. We can use this approach to show equalities from Section 3: for example, to prove associativity of ++. i.e., `(xs, ys, zs) => (xs ++ ys) ++ zs ≡ (xs, ys, zs) => xs ++ (ys ++ zs)`, we can transform both sides to:

```
fix(f => (xs, ys, zs) => xs is of [] => ys ++ zs
                              of u::us => u::(f us ys zs))
```

This approach is at times useful even for deforestation. In particular, it lets us reproduce the motivating example of superfusion [17], which deforests `(ls) => max(map(sum, tails(ls)))` into a linear version that keeps a pair accumulator of current max of tail sums along with the sum of the current tail. Superfusion uses program synthesis to achieve this result, while we reproduce it using only our small language of tactics.

At times when using this meet-in-the-middle approach the difference between the transformed left-hand side and the desired right-hand side is quite small, e.g., like in the associativity example above. We hypothesize that fixpoint-fusion inspired tactics could more elegantly handle such small cuts [32].

Pattern-match exhaustiveness The pattern matches in the $\mathcal{L}$ language may not be exhaustive. In the absence of matching patterns, an expression can reduce to a stuck term, '`s is`', without any branches. Unfortunately, this property prevents us from eliminating redundant matches whose branches all share the same body. This limitation could be addressed by adding a notion of types, allowing us to check exhaustiveness of the match, or by also tracking refinements in addition to equalities. We use the (unsafe) collapse-branches tactics to deforest the example from the superfusion paper [17].

Call-by-value semantics By imposing a termination precondition on the deforested terms, we make sure that a deforestation script cannot introduce non-termination. This property would not hold in call-by-value semantics. Consider the term `if [] = [] then 1 else ((x) => x)((x) => x)`, which reduces to 1: if we perform abstraction on this term, we rewrite it to `((e) => if [] = [] then 1 else e)(((x) => x)((x) => x))`, and thus introduce nontermination. Previous work [8] addresses this issue by thunking.

Implementation We have created a prototype implementation[2] of this deforestation system in Scala. The implementation is a REPL where one can load the global context from chosen files, create a goal, and deforest it using our tactics. Tactics like PMSubs have options that allow specifying which terms should be substituted. All arguments and alpha-renames for Fold and FoldFix are explicitly passed, similarly for Rewrite. The Focus tactic exists in two flavors – one for single transformations, and one for sequences of transformations. The first option has a short-hand notation for combining it with other tactics, while the second temporarily shelves the current goal, creating a new one with the chosen subtree as the subject. For this tactic we specify the subterm using its *path*. However, for tactics like Abstract, where we may want to abstract over more than a single occurrence of a subterm, we need to pass the subterm by writing it out, and optionally pass numbers of occurrences that should be abstracted over (if not all). The implementation also differs in some details from the formalization given here: there are more language constructs, e.g., boolean literals; we support multiple-argument abstractions instead of currying automatically; and to use the Fix operation we require the initial definition to be bound. In the implementation we also have commands that allow one to get information about the current context, inspect the paths for the subterms, and, most interestingly, one that allows one to create custom, unchecked equalities.

5 Related work

Deforestation Our work inherits from a rich literature on program transformation [7], including the fold/unfold approach [27], partial evaluation with narrowing [2, 1], and deforestation [37], including shortcut fusion [12, 33, 23], supercompilation [34, 35], and distillation [14]. Most of these systems develop bespoke algorithms rather than programs in a small DSL (our second objective). Some of these systems allow for user extensions (GHC, for example, allows users to add new rewrite rules [28]), but few of them give control on the results rather than the process (our third objective). Superfusion [17] is one exception: users can mark a data structure as `Packed` to indicate that it should be eliminated, which gives a way to express simple deforestation objectives but not more complex ones such as the intention

[2] A web version of our prototype can be accessed at `https://systemf.epfl.ch/etc/eco/`.

to fuse two maps. Another exception are stream-fusion languages [9]: in exchange for a limited API, *complete* stream fusion [19, 20] guarantees that *all* intermediate structures are removed. Our work is also less tightly connected to newer approaches to deforestation, like type-based deforestation [8], where type inference is used for automatically finding fusion opportunities.

Programmer-guided and interactive optimization User-driven optimization using transformation scripts and rewriting strategies is equally well established [5, 30, 29, 25, 3, 13, 16, 21, 22, 24]. However, none of these efforts focus explicitly on deforestation, and most do not allow user-supplied optimization objectives as a way to mix automation and user control. Still, their success suggests that user guidance is practical: we envision a system integrating a transformation language in the compiler in the style of KURE, HERMIT [10, 31, 11], and Stratego [6], offering a minimal set of deforestation rules and control structures, and controlling the tradeoff between automation and handwritten scripts using annotations to pick strategies and set objectives.

References

[1] Elvira Albert, Michael Hanus & Germán Vidal (2000): *Using an Abstract Representation to Specialize Functional Logic Programs*. In Michel Parigot & Andrei Voronkov, editors: *Logic for Programming and Automated Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 381–398.

[2] Elvira Albert & German Vidal: *The narrowing-driven approach to functional logic program specialization* 20(1), pp. 3–26. doi:10.1007/BF03037257. Available at `https://doi.org/10.1007/BF03037257`.

[3] João Bispo & João M. P. Cardoso (2020): *Clava: C/C++ source-to-source compilation using LARA*. *SoftwareX* 12, p. 100565, doi:10.1016/j.softx.2020.100565. Available at `https://www.sciencedirect.com/science/article/pii/S2352711019302122`.

[4] Maximilian Bolingbroke & Simon Peyton Jones (2010): *Supercompilation by evaluation*. In: *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, Association for Computing Machinery, pp. 135–146, doi:10.1145/1863523.1863540. Available at `https://dl.acm.org/doi/10.1145/1863523.1863540`.

[5] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas & Eelco Visser (2008): *Stratego/XT 0.17. A language and toolset for program transformation*. *Science of Computer Programming* 72(1), pp. 52–70, doi:10.1016/j.scico.2007.11.003. Available at `https://www.sciencedirect.com/science/article/pii/S0167642308000452`.

[6] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas & Eelco Visser (2008): *Stratego/XT 0.17. A language and toolset for program transformation*. *Science of Computer Programming* 72(1), pp. 52–70, doi:https://doi.org/10.1016/j.scico.2007.11.003. Available at `https://www.sciencedirect.com/science/article/pii/S0167642308000452`. Special Issue on Second issue of experimental software and toolkits (EST).

[7] R. M. Burstall & John Darlington (1977): *A Transformation System for Developing Recursive Programs*. *Journal of the ACM* 24(1), pp. 44–67, doi:10.1145/321992.321996. Available at `http://dx.doi.org/10.1145/321992.321996`.

[8] Yijia Chen & Lionel Parreaux (2024): *The Long Way to Deforestation: A Type Inference and Elaboration Technique for Removing Intermediate Data Structures*. *Proceedings of the ACM on Programming Languages* 8(ICFP), pp. 249–283, doi:10.1145/3674634. Available at `https://dl.acm.org/doi/10.1145/3674634`.

[9] Duncan Coutts, Roman Leshchinskiy & Don Stewart (2007): *Stream fusion: from lists to streams to nothing at all*. *SIGPLAN Not.* 42(9), pp. 315–326, doi:10.1145/1291220.1291199. Available at `https://dl.acm.org/doi/10.1145/1291220.1291199`.

[10] Andrew Farmer, Andy Gill, Ed Komp & Neil Sculthorpe (2012): *The HERMIT in the machine: a plugin for the interactive transformation of GHC core language programs*. In: *Proceedings of the 2012 ACM*

SIGPLAN Symposium on Haskell, Haskell '12, ACM, pp. 1–12, doi:10.1145/2364506.2364508. Available at http://dx.doi.org/10.1145/2364506.2364508.

[11] Andrew Farmer, Neil Sculthorpe & Andy Gill (2015): *Reasoning with the HERMIT: tool support for equational reasoning on GHC core programs*. In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, ACM, pp. 23–34, doi:10.1145/2804302.2804303. Available at http://dx.doi.org/10.1145/2804302.2804303.

[12] Andrew Gill, John Launchbury & Simon L. Peyton Jones (1993): *A short cut to deforestation*. In: *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, Association for Computing Machinery, pp. 223–232, doi:10.1145/165180.165214. Available at https://dl.acm.org/doi/10.1145/165180.165214.

[13] Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, Xueying Qin, Sergei Gorlatch & Michel Steuwer (2020): *Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies*. *Proceedings of the ACM on Programming Languages* 4(ICFP), pp. 1–29, doi:10.1145/3408974. Available at http://dx.doi.org/10.1145/3408974.

[14] G. W. Hamilton (2007): *Distillation: extracting the essence of programs*. In: *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, Association for Computing Machinery, pp. 61–70, doi:10.1145/1244381.1244391. Available at https://dl.acm.org/doi/10.1145/1244381.1244391.

[15] Geoffrey Hamilton: *The Next 700 Program Transformers*, doi:10.48550/arXiv.2108.11347. Available at http://arxiv.org/abs/2108.11347.

[16] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc & Jonathan Ragan-Kelley (2022): *Exocompilation for productive programming of hardware accelerators*. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, Association for Computing Machinery, New York, NY, USA, p. 703–718, doi:10.1145/3519939.3523446. Available at https://doi.org/10.1145/3519939.3523446.

[17] Ruyi Ji, Yuwei Zhao, Nadia Polikarpova, Yingfei Xiong & Zhenjiang Hu (2024): *Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis*. *Proc. ACM Program. Lang.* 8(PLDI), pp. 939–964, doi:10.1145/3656415. Available at https://dl.acm.org/doi/10.1145/3656415.

[18] Manohar Jonnalagedda & Sandro Stucki: *Fold-based fusion as a library: a generative programming pearl*. In: *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*, SCALA 2015, Association for Computing Machinery, pp. 41–50, doi:10.1145/2774975.2774981. Available at https://dl.acm.org/doi/10.1145/2774975.2774981.

[19] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos & Yannis Smaragdakis (2017): *Stream fusion, to completeness*. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, Association for Computing Machinery, New York, NY, USA, pp. 285–299, doi:10.1145/3009837.3009880. Available at https://doi.org/10.1145/3009837.3009880.

[20] Tomoaki Kobayashi & Oleg Kiselyov (2024): *Complete Stream Fusion for Software-Defined Radio*. In: *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, PEPM 2024, Association for Computing Machinery, New York, NY, USA, pp. 57–69, doi:10.1145/3635800.3636962. Available at https://doi.org/10.1145/3635800.3636962.

[21] Thomas Koehler, Arthur Charguéraud, Begatim Bytyqi, Damien Rouhling & Yann A Barsamian (2023): *OptiTrust: an Interactive Optimization Framework*. Available at https://inria.hal.science/hal-04053772. Published: ARRAY 2023 - Workshop - PLDI 2023.

[22] Thomas Kœhler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder & Michel Steuwer (2024): *Guided Equality Saturation*. *Proceedings of the ACM on Programming Languages* 8(POPL), pp. 1727–1758, doi:10.1145/3632900. Available at http://dx.doi.org/10.1145/3632900.

[23] John Launchbury & Tim Sheard: *Warm fusion: deriving build-catas from recursive definitions*. In: *Proceedings of the seventh international conference on Functional programming languages and computer architec-*

ture, FPCA '95, Association for Computing Machinery, pp. 314–323, doi:10.1145/224164.224223. Available at https://dl.acm.org/doi/10.1145/224164.224223.

[24] Martin Paul Lücke, Oleksandr Zinenko, William S. Moses, Michel Steuwer & Albert Cohen (2025): *The MLIR Transform Dialect: Your Compiler Is More Powerful Than You Think*. In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, ACM, pp. 241–254, doi:10.1145/3696443.3708922. Available at http://dx.doi.org/10.1145/3696443.3708922.

[25] Kedar S. Namjoshi & Nimit Singhania (2016): *Loopy: Programmable and Formally Verified Loop Transformations*. In Xavier Rival, editor: *Static Analysis*, Springer, pp. 383–402, doi:10.1007/978-3-662-53413-7_-19.

[26] Dmitry Petrashko, Ondřej Lhoták & Martin Odersky: *Miniphases: compilation using modular and efficient tree transformations*. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, Association for Computing Machinery, pp. 201–216, doi:10.1145/3062341.3062346. Available at https://dl.acm.org/doi/10.1145/3062341.3062346.

[27] Alberto Pettorossi & Maurizio Proietti (1996): *Rules and strategies for transforming functional and logic programs*. *ACM Comput. Surv.* 28(2), p. 360–414, doi:10.1145/234528.234529. Available at https://doi.org/10.1145/234528.234529.

[28] Simon Peyton Jones, Andrew Tolmach & Tony Hoare (2001): *Playing by the rules: rewriting as a practical optimisation technique in GHC*. In: *Proceedings of the 2001 Haskell Workshop*. Available at https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/.

[29] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand & Saman Amarasinghe (2013): *Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines*. *SIGPLAN Not.* 48(6), p. 519–530, doi:10.1145/2499370.2462176. Available at https://doi.org/10.1145/2499370.2462176.

[30] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen & Jacqueline Chame (2011): *A Programming Language Interface to Describe Transformations and Code Generation*. In Keith Cooper, John Mellor-Crummey & Vivek Sarkar, editors: *Languages and Compilers for Parallel Computing*, Springer, pp. 136–150, doi:10.1007/978-3-642-19595-2_10.

[31] Neil Sculthorpe, Nicolas Frisby & Andy Gill (2014): *The Kansas University rewrite engine: A Haskell-Embedded Strategic Programming Language with Custom Closed Universes*. *Journal of Functional Programming* 24(4), pp. 434–473, doi:10.1017/S0956796814000185.

[32] William Sonnex (2017): *Fixed point promotion: taking the induction out of automated induction*. Technical Report UCAM-CL-TR-905, University of Cambridge, Computer Laboratory, doi:10.48456/tr-905. Available at https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-905.html.

[33] Josef Svenningsson (2002): *Shortcut fusion for accumulating parameters & zip-like functions*. In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, Association for Computing Machinery, pp. 124–132, doi:10.1145/581478.581491. Available at https://dl.acm.org/doi/10.1145/581478.581491.

[34] M. H. Sørensen, R. Glück & N. D. Jones (1996): *A positive supercompiler*. *Journal of Functional Programming* 6(6), pp. 811–838, doi:10.1017/S0956796800002008. Available at https://www.cambridge.org/core/journals/journal-of-functional-programming/article/positive-supercompiler/4EEE2EBC972AA2FDC861EF7A713EE898.

[35] Morten Heine B. Sørensen & Robert Glück (1999): *Introduction to Supercompilation*. In John Hatcliff, Torben Æ Mogensen & Peter Thiemann, editors: *Partial Evaluation*, Springer, pp. 246–270, doi:10.1007/3-540-47018-2_10.

[36] Valentin F. Turchin (1986): *The concept of a supercompiler*. *ACM Trans. Program. Lang. Syst.* 8(3), pp. 292–325, doi:10.1145/5956.5957. Available at https://dl.acm.org/doi/10.1145/5956.5957.

[37] Philip Wadler (1990): *Deforestation: transforming programs to eliminate trees*. *Theoretical Computer Science* 73(2), pp. 231–248, doi:10.1016/0304-3975(90)90147-A. Available at `https://www.sciencedirect.com/science/article/pii/030439759090147A`.