Oracle-based Differential Operational Semantics (long version)

Thibaut Girka, David Mentré, Yann Régis-Gianas

Oracle-based Differential Operational Semantics
(long version)

Thibaut Girka[1,2], David Mentré[1], and Yann Régis-Gianas[2]

[1]Mitsubishi Electric R&D Centre Europe, F-35708 Rennes, France
[2]Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, PiR2, INRIA Paris-Rocquencourt, F-75205 Paris, France

Abstract. Program differences are pervasive in software development and understanding them is crucial. However, such changes are usually represented as textual differences with no regard to the syntactic nature of programs or their semantics. Such a representation may be hard to read or reason about and often fails to convey insight on the semantic implications of a change. In this paper, we propose a formal framework to characterize the difference of behavior between two close programs—equivalent or not—in terms of their small-step semantics. To this end, we introduce small-step-prediction oracles that consume one reduction step of one program and produce a sequence of reduction steps of the other. Such oracles are operational, handle diverging or stuck computations, and are well-suited for describing local changes, while expressive enough to describe arbitrary ones. They can also be composed, to characterize a difference as a sequence of simpler differences. Last but not least, small-prediction-step oracles can be explained to programmers in terms of evaluation of the compared programs. We illustrate this framework by instantiating it on the Imp imperative language, with oracles ranging from trivial equivalence-preserving syntactic transformations to characterized semantic differences. Through these examples, we show how our framework can be used to relate syntactic changes with their effect on the semantics, or to describe higher-level changes by abstracting away from the small-step semantics presentation. We have defined and proved the framework and the presented examples in the Coq proof assistant, and implemented a proof-of-concept inference tool for the Imp language.

1 Introduction

1.1 A foundational framework for semantic differences

How should we formally describe and characterize the difference between the behaviors of two close programs? This question encompasses the concerns of both programmers and semanticists.

In software development, this question is essential for the programmer to understand the impact of a change made on a program source code: Does this change introduces a bug or, on the contrary, does it fix one? Does it preserve the semantics of the original program? Does it extend or restrict the features

of the original program? In the current practice, programmers compare two close programs using textual differences between their source codes. These textual differences are manually reviewed, applied as patches, composed, compared, merged and versioned. But, because textual differences are both low-level and unstructured, none of these operations have a clear characterization in terms of program semantics and therefore the tools that implement these operations cannot be trusted. The programmer has to manually interpolate the impact of some textual change to understand if these operations are licit. This interpolating proccess is error-prone, especially on large software. What if the programmer was given semantically grounded and high-level *difference languages*? Using these languages, the intent behind patches could be expressed, mechanically checked and reviewed in a high-level and semantic-aware setting, paving the way for *certified evolution* of software.

The working semanticist must answer this question when studying program transformations that *do not preserve the semantics* of the source program. Unfortunately, in that situation, the semanticist is not well equipped. Indeed, while there are plenty of foundational frameworks to deal with equivalence between programs, very few attempts [1, 11, 3, 18] have been made to provide general frameworks to characterize how two close programs differ semantically. The versatility of the notion of closedness between programs may explain this: saying something relevant about two programs that compute differently has no general answer and mainly depends on the application. What if the semanticist learnt how to formally devise difference languages and prove meta-properties about them? These difference languages, defined in a common logic, could be helpful to formally express relations between program traces that are not equivalent, to measure edition distance between programs or to prove the soundness of differential static analyses [24, 21, 27, 18], impact analyses or automatic bug-fixing process.

1.2 Differential operational semantics

In this paper, we propose a foundational framework—implemented in the Coq proof assistant—to define difference languages. The design of this framework has been highly inspired by the seminal work of Plotkin on operational semantics [20]. An operational semantics defines a set of syntactical rules to assign meaning to a program written in a programming language $\mathcal{L}$. In other words, an operational semantics defines a program—an interpreter—that maps a program to its behavior (be it a single value or a potentially infinite chain of reduction steps). By analogy, a differential operational semantics defines a program—a difference interpreter—that maps a difference between two programs to a relation between the behaviors of these two programs.

For a given programming language, there may be a lot of useful difference languages. A difference language may characterize pairs of programs that are semantically equivalent *modulo* some implementation details: such difference languages capture pairs of programs that are related by a renaming of their variables, that produce the same results but using different algorithms, that

reorder independent instructions, etc... A difference language may also characterize pairs of programs *that are not semantically equivalent but whose difference of behaviors falls into well-defined categories*: that kind of difference languages captures pairs of programs (P_1, P_2) for which P_2 includes the execution traces of P_1 but has also other behaviors (program refinement, features extension or bug fixing), for which P_2 and P_1 interpret their inputs similarly but differ on the way they produce their outputs, etc...

In this work, we present basic properties about semantic differences and, when possible, we prove that these properties hold for all the differences written in specific difference languages. Amongst these properties, we characterize the differences that are (i) sound for a given pair of programs, (ii) applicative, and (iii) that can be composed together while still remaining informative. To ease the mechanization of this metatheoretical work, we follow an operational approach explained in the next paragraphs.

1.3 Small-prediction-steps oracles

Just like any standard interpreter, a difference interpreter defines the meaning of its input in a computational way. Even if any computable function may be used to implement a difference interpreter, we chose to restrict ourselves to difference interpreters that follow a discipline that we call *small-prediction-steps*: roughly speaking, at each interpretation step, the interpretation of a difference δ between P_1 and P_2 must consume one reduction step of P_1 or P_2 and produce *a (potentially empty) finite sequence of reduction steps of the other program.* In other words, the interpretation of a difference acts as a *difference oracle* $\mathcal{O}(\delta)$ predicting at each step the behavior of one program given the behavior of the other. Hence, the relation between the two program behaviors is computed along the two evaluations of these programs as depicted by the following diagram:

$$\begin{array}{lccccccc}
P_1 & c_{11} & \xrightarrow{\delta_1} & c_{12} & \dashrightarrow & c_{13} & \cdots \\
 & \vert & \Vert & \vert & \Uparrow & \vert & \\
 & \sim & \mathcal{O}(\delta_1) & \sim & \mathcal{O}(\delta_2) & \sim & \\
 & \vert & \Downarrow & \vert & \Vert & \vert & \\
P_2 & c_{21} & \dashrightarrow & c_{22} & \xrightarrow[\delta_2]{} & c_{23} & \cdots
\end{array}$$

In this diagram, the plain arrows ($\rightarrow$) correspond to execution steps of one program given as input to the oracle while the dashed arrows ($\dashrightarrow$) correspond to the oracle prediction of zero, one or several execution steps of the other program. The vertical lines correspond to the relation between the program configurations built along the execution of the oracle. In the sequel though, we will not mention the oracle explicitly as in this diagram since the nature of the arrows is enough to determine the direction in which the oracle has been used.

Like small-step operational semantics, they can handle diverging or stuck computations. Difference languages can be made precise enough to distinguish programs that only differ on a single instruction ; or on the contrary, abstract enough to equate extensionally equivalent but distinct algorithms. Besides, a small-prediction-step can be made local in the sense that it does not depend

on the evaluation contexts of the two programs it compares. This property is important because if accumulated along multiple iterations, a small simple difference between two versions of a program may become complex, as shown in the following example:

```
1 while (1) {
2   x = x + 1;
3 }
```

```
1 while (1) {
2   x = x * x + 1;
3 }
```

To characterize the differences between the states of the two programs at each loop iteration, it is more convenient to say that each time P_1 increments x, P_2 squares and increments x than trying to find a general term for the differences between the values of x in the two programs.

Small-prediction-steps oracles have another nice property: they can be composed. This is important because two (even seemingly close) programs may differ in various and complex ways depending on what we are observing on their operational behavior and depending on how close the two program traces are. To deal with this complexity, a complex semantic difference between two programs should be decomposable into a sequence of atomic semantic differences.

Finally, small-prediction-steps oracles can be informally explained to programmers: a precise enough informal documentation of a difference language can be expressed in terms of the evaluations of compared programs, which are usually well apprehended by programmers.

Contributions The first contribution of this paper is to introduce a notion of differential operational semantics based on small-prediction-steps oracles in a language agnostic way. For the clarity of exposition, we instantiate the framework on a standard toy imperative language Imp equipped with arithmetic expressions, mutable variables, conditional statements and unbounded iterations. Yet, our general definitions only assume that the programming language is deterministic. (We come back on this restriction to deterministic languages in Section 5.2.) Several examples of difference languages on Imp are introduced and several of their properties are proved along the presentation.

The second contribution of this paper is to classify difference languages and small-prediction-steps oracles with respect to the metaproperties they enjoy: checking if an oracle realizes a valid semantic difference between two programs may or may not be decidable for a given difference language ; inferring a valid oracle to relate two programs may also be undecidable; the composition of two oracles is possible under some specific conditions.

As a third contribution, a prototype implementation of the framework has been developed using the Coq proof assistant. It contains the language-independent definitions as module functors and the difference languages presented in this paper (both definitions, theorems and their proofs). The framework is already able to serve as a target for the certificates produced by a toy semantic difference engine we developed in OCaml for the Imp programming language. A companion

technical report, the full Coq development and the OCaml tool can be found online[1] along with examples.

Outline In Section 2, we illustrate that the framework can characterize the difference between two close programs. Section 3 introduces a change-based presentation of standard small-step operational semantics on top of which the general definitions for difference languages and small-prediction-steps oracles are built in Section 4. The metaproperties about difference languages are introduced in Section 5. We focus on the properties needed to compose oracles in Section 6. Section 7 describes the implementation of a semantic difference engine for Imp. Related work is discussed in Section 8 and our research agenda in Section 9.

Preliminaries The formal development of this paper have been conducted in Type Theory but, we will sometimes abusively use the vocabulary of mathematics based on Set Theory in our statements and definitions. For instance, the expression "partial function" from A to B, written $A \nrightarrow B$, must be understood as a function of type "$A \to$ option B". Similarly, we use the word "subset of A" to denote a predicate of type $A \to$ Prop. When some inductive type A defines a set of first-order terms, its definition will be given by a BNF grammar and we will use A both as a type and as a metavariable ranging over terms of this type. We also write $\overline{A}$ to range over vectors of A. Most of the time, functions over A are silently lifted to $\overline{A}$ when the context suffices to deduce how.

2 Overview of differential oracles

In this section, we informally illustrate how the framework of differential operational semantics can be used to relate two imperative program fragments. The two following programs come from the gallery of verified programs of Why3 [4].

```
sum = 0;
x = -x;
y = 0;
while (sum < x) {
  y = y + 1;
  sum = sum + 1 + 2 * y;
}
sum = 0;
```

Original program P_0

```
x = -x;
count = 0;
if (x < 4)  { count = 1; }
else {
  count = x;
  sum = (x + 1) / 2;
  while (sum < count) {
    count = sum;
    sum = ((x / sum) + sum) / 2;
  }
}
sum = 0;
```

Modified program P_3

The first program P_0 computes the square root of the opposite of a negative integer x using a simple iterative algorithm that looks for the integer y such that $y^2 \leq (-x) < (y+1)^2 = y^2 + 2y + 1$. The second program P_3 also computes the square root of $-x$ but using a Newton-Raphson-like root searching algorithm.

[1] https://www.irif.fr/~thib/oracles/

Contrary to P_0, P_3 stores the final result into the variable `count` rather than into the variable `y`.

In both programs, the variable `sum` is set to zero at the end of the square root calculation to simulate a form of variable scoping in the IMP programming language. Indeed, the variable `sum` is morally a temporary variable and it should not intervene in the comparison of the final results. By setting the same value to `sum` in both programs, we force the two final memories to be compared only with respect to the variable containing the square root.

To formally relate P_0 and P_3, we introduce in Figure ?? two intermediate programs P_1 and P_2 such that for each i, there exist a difference δ_i written in a well-chosen difference language Δ^i_{IMP} that characterizes the difference between the behaviors of P_i and P_{i+1}. The difference between P_0 and P_3 is $\bigcirc_{i=0}^{2}\delta_i$, *i.e.* the composition of all these differences.

```
1 sum = 0;
2 x = -x;
3 count = 0;
4 while (sum < x) {
5   count = count + 1;
6   sum = sum + 1 + 2 * count;
7 }
8 sum = 0;
```

Intermediate program P_1

```
1 x = -x;
2 sum = 0;
3 count = 0;
4 while (sum < x) {
5   count = count + 1;
6   sum = sum + 1 + 2 * count;
7 }
8 sum = 0;
```

Intermediate program P_2

Variable renaming Variable renamings is one of the simplest difference languages. In this language, a difference is denoted by a bijection between the variables of the two programs. In our example, to have P_0 and P_3 agree on the variable used to store the square root, we use P_1 which is related to P_0 by a difference written "rename `y` $\leftrightarrow$ `count`" in the language $\Delta^{Ren}_{\text{IMP}}$ (defined in Section 4.2). The semantics of this difference is the oracle that renames, for each execution step of P_0, the identifier `y` into `count` in the configuration of P_0 to produce the predicted configuration for P_1. This difference is sound because the predicted configuration effectively corresponds to the next execution step of P_1.

Notice that in $\Delta^{Ren}_{\text{IMP}}$, the traces of P_0 and P_1 correlated by the oracle are *synchronized*: for each input step, a single step is predicted. Besides, the oracle is *bidirectional* because a renaming between P_0 and P_1 can be interpreted both ways: to predict a step of P_1 from a step of P_0 or to predict a step of P_0 from a step of P_1.

Assignment commutation The programs P_1 and P_3 are still quite different but one can notice that they both start with variable initializations. More precisely, the first instruction P_3 initializes `x` while this initialisation only happens in the second instruction of P_1. To get a bit closer to P_3 we introduce P_2 which is related to P_1 by the difference "swap assignments at line 1" from the language $\Delta^{SwapAssign}_{\text{IMP}}$ (defined in Section 4.2). For the semanticist, the oracle for this difference language is a bit more difficult to define than the previous one since it cannot return one prediction for each execution step of P_1: after the execution of the first assignment of P_1, the oracle cannot provide a valid

prediction yet because there is no configuration of P_2 in which `sum` is initialized but `x` is not. Hence, the oracle must *wait* for another step of P_1 before returning the state of P_2 that results in the execution of the first two instructions.

When it relates two assignments in opposite order in the two programs, the oracle is desynchronized: it may predict zero, one or two execution steps of one program after it has consumed one execution step of the other program. Furthermore, the oracle is not bidirectional in that case: when it is between two commuted assignments predicting P_2 from P_1, it cannot be asked to predict P_1 from P_2.

Abstraction After the initialization of `count`, the difference between P_2 and P_3 is difficult to express if we stay at the level of a single execution step because the algorithm to compute the square root is different in each program. This situation illustrates one powerful aspect of differential operational semantics: a difference language can be used to abstract away irrelevant implementation details. An oracle can skip one of the two algorithms and relate extensionnally the states of the two programs as found after the execution of the two algorithms.

The difference language $\Delta_{\text{IMP}}^{AbstractEquiv}$ (defined in Section 4.2) captures such high-level program comparison. In our example, the program P_2 would be related to P_3 by the difference written "abstract equivalence between lines 2−8 and lines 2−12". In that case, the oracle abstracts away the instructions of P_3 from line 2 to line 12 by a single prediction asserting that the configuration of P_3 will be equivalent to the configuration of P_2 when it reaches line 12. This prediction is valid only if there is a proof that these two algorithms are equivalent and that a bound on the number of steps needed to skip the instructions is given. More generally, the soundness of an oracle can be conditioned by proof obligations.

Note that we could have avoided describing the previous change since $\Delta_{\text{IMP}}^{SwapAssign}$ is subsumed by $\Delta_{\text{IMP}}^{AbstractEquiv}$. In general, the difference language $\Delta_{\text{IMP}}^{AbstractEquiv}$ is a sledgehammer solution to express differences between equivalent programs, but requires the user to provide proofs, and may not relate intermediate states of the two program's execution states as precisely as specialized oracle languages.

Putting it all together In the end, a sound difference between P_0 and P_3 is obtained by the following *composition* of the atomic differences we explicited:

```
1  rename y ↔ count;
2  swap assignments at line 1;
3  abstract equivalence between lines 2-8 and lines 2-12.
```

One can imagine this concise difference to be written by a programmer as a formal description of their intent, to be checked by a differential static analyser, or even to be inferred by a semantic comparison tool. In any case, the well-defined semantics of difference languages make them amenable to an integration to development and certification tools.

The semantics of this composition is given by an oracle which is obtained by composing the prediction of the oracles of each atomic difference. As the oracle for "swap assignments at line 1" is not always bidirectional, the composed

oracle cannot be always bidirectional either. More generally, the composed oracle inherits the directionality constraints from the oracles it is composed of.

3 Change-indexed operational semantics

As coined in the introduction, difference languages are built on top of change-indexed presentations of small-step operational semantics. This presentation of small-step operational semantics is extensionally equivalent to the usual ones except that it describes the relation between program configurations more precisely, intentionally speaking. Indeed, a standard small-step relation usually written "$c_1 \to c_2$" is a binary relation between a configuration c_1 before the evaluation step and a configuration c_2 after the evaluation step. In comparison, a change-indexed small-step relation describes *how* c_2 has been produced from c_1 using a change d taken in a so-called "change structure" [6] such that "$c_1 \oplus d = c_2$". In other words, the purpose of change-indexed operational semantics is to reify execution steps as first-class values so that they can be given as input to oracles. In Section 3.1, we formally define change structures and change-indexed small-step operational semantics. These presentations of operational semantics are necessary to write context-independent oracles. Section 3.2 contains the change-indexed small-step operational semantics for Imp.

3.1 Change-based interpretation of programs

Change structure For a given value v of type A, a change dv over A is a value of type ΔA that can be applied to v to get a new value of type A using the change application operator $\oplus$. Two changes can be composed using the operator $\odot$.

Definition 1. *Given a type A, a change structure over A is a 4-uple $(\Delta A, \oplus, \odot, 0)$ such that:*

- *ΔA is the type for change and we have $0 : \Delta A$; $\oplus : A \to \Delta A \to A$ and $\odot : \Delta A \to \Delta A \to \Delta A$;*
- *$\forall x : A, x \oplus 0 = x$;*
- *$\forall x : A, d_1, d_2 : \Delta A, (x \oplus d_1) \oplus d_2 = x \oplus (d_1 \odot d_2)$.*

This definition of change structure differs from Cai et al's in several aspects. First, in the original formulation of change structure, a change for a value v has a dependent type Δv. We use a simple type ΔA instead. As a consequence, a well-typed application of a change to a value can be undefined. We will simply make sure that this case never happens in our inference rules. Second, the original definition of a change structure includes a substraction operator $\ominus$ such that $v \ominus u$ is a change from u to v. We do not need such an operation in our framework. Third, we introduce the operator $\odot$ which allows the composition of changes. This is required to represent the effect of several reduction steps on a configuration.

$$
\begin{array}{l}
C ::= \textbf{skip} \mid x = e \mid C; C \mid \textbf{if}\ (b)\ C\ \textbf{else}\ C \mid \textbf{while}\ (b)\ C \\
e ::= n \mid x \mid e + e \mid e * e \mid e - e \mid e/e \\
b ::= \textbf{true} \mid \textbf{false} \mid !b \mid b\ \&\&\ b \mid b \mid\mid b \mid e = e \mid e \leq e \\
c ::= (M, \kappa) \quad M ::= \bullet \mid M[x := n] \quad \kappa ::= \textbf{halt} \mid C; \kappa
\end{array}
$$

Fig. 1. Syntax of Imp.

Change-indexed reduction rules A programming language $\mathcal{L}$ is a set of program source codes. A (deterministic) small-step operational semantics is defined by a set of configurations $\mathcal{C}$, a subset $\mathcal{I}$ (resp. $\mathcal{F}$) of initial (resp. final) configurations and a partial function step from $\mathcal{C}$ to $\mathcal{C}$ such that $\mathsf{step}(c)$ is the configuration reached from c after a single execution step. We assume that there is a unique initial configuration for each program. So, we can write $\mathcal{I}(P)$ to denote the initial configuration of program P.

Definition 2. *The* change-indexed *presentation of a small-step operational semantics* $(\mathcal{C}, \mathcal{I}, \mathcal{F}, \mathsf{step})$ *is a change structure over* $\mathcal{C}$ *and a partial function* $\Delta\mathsf{step}$ *from* $\mathcal{C}$ *to* $\Delta\mathcal{C}$ *such that* $\forall c, \mathsf{step}(c) = c \oplus \Delta\mathsf{step}(c)$. *The configuration is* stuck *when* $\Delta\mathsf{step}(c)$ *is undefined.*

3.2 Change-based interpretation of Imp programs

A program in the Imp programming language is written using the standard syntax of commands and expressions as described in Figure 1. We consider the small-step operational semantics of Imp whose configurations c are pairs of a store M and a continuation κ. For every program C, the configuration $(\bullet, C; \textbf{halt})$ is initial and for every store M, the configuration $(M, \textbf{halt})$ is final. The step function is defined as usual: we omit the obvious definitions as well as expressions evaluation rules. (They can be find in appendix though.)

There exist many change structures over Imp configurations. We choose the syntax for changes over continuations, stores and configurations described in Figure 2. The semantics of these changes is specified by the action of $\oplus$ on their corresponding values. The function $\Delta\mathsf{step}$ simply reifies the change made by the standard evaluation function on the configurations. The change **pop** removes the current command (the one at the top of the continuation). The changes starting with u represents all the transformations of the current command that can occur during evaluation.

In the sequel, we will use the standard notation "$c_1 \to c_2$" when the change is not relevant to the context. We will also write $c_1 < c_2$ if c_2 appears after c_1 in the reduction chain or if c_1 and c_2 are both final configurations.

Syntax for changes

$$\begin{aligned}\delta\kappa &::= \mathbf{pop} \mid u\mathbf{seq} \mid u\mathbf{then} \mid u\mathbf{else} \mid u\mathbf{while}\\ \delta M &::= x := n\\ \delta C &::= (\overline{\delta\kappa}, \overline{\delta M})\end{aligned}$$

Change semantics

$$\begin{aligned}C;\kappa \oplus \mathbf{pop} &= \kappa\\ (C_1;C_2);\kappa \oplus u\mathbf{seq} &= C_1;(C_2;\kappa)\\ \mathbf{if}\ (b)\ C_1\ \mathbf{else}\ C_2;\kappa \oplus u\mathbf{then} &= C_1;\kappa\\ \mathbf{if}\ (b)\ C_1\ \mathbf{else}\ C_2;\kappa \oplus u\mathbf{else} &= C_2;\kappa\\ \mathbf{while}\ (b)\ C;\kappa \oplus u\mathbf{while} &= C;\mathbf{while}\ (b)\ C;\kappa\\ M \oplus x := n &= M[x := n]\end{aligned}$$

Change-indexed operational semantics

$$\begin{aligned}\Delta\mathsf{step}(M, \mathbf{skip};\kappa) &= (\epsilon, \mathbf{pop})\\ \Delta\mathsf{step}(M, x = e;\kappa) &= (x := n, \mathbf{pop}) && \text{where } M \vdash e \Downarrow n\\ \Delta\mathsf{step}(M, (C_1;C_2);\kappa) &= (\epsilon, u\mathbf{seq})\\ \Delta\mathsf{step}(M, \mathbf{if}\ (b)\ C_1\ \mathbf{else}\ C_2;\kappa) &= (\epsilon, u\mathbf{then}) && \text{where } M \vdash b \Downarrow \mathbf{true}\\ \Delta\mathsf{step}(M, \mathbf{if}\ (b)\ C_1\ \mathbf{else}\ C_2;\kappa) &= (\epsilon, u\mathbf{else}) && \text{where } M \vdash b \Downarrow \mathbf{false}\\ \Delta\mathsf{step}(M, \mathbf{while}\ (b)\ C;\kappa) &= (\epsilon, u\mathbf{while}) && \text{where } M \vdash b \Downarrow \mathbf{true}\\ \Delta\mathsf{step}(M, \mathbf{while}\ (b)\ C;\kappa) &= (\epsilon, \mathbf{pop}) && \text{where } M \vdash b \Downarrow \mathbf{false}\end{aligned}$$

Fig. 2. Change-based presentation of Imp semantics

4 Difference languages

4.1 How to define a difference language?

From a change-based presentation of the semantics of a programming language, the semanticist can define a difference language. As usual, the formalization of a language is made of a syntax and a semantics for the terms of this syntax. The specificity of difference languages is their interpretation function which maps a difference to its *prediction function*, which itself realizes the so-called difference oracle. In this section, we first explain the type of prediction functions, how they interact with converging, diverging and stuck programs and we finally give the formal definition of what a difference language is.

What is the type of a prediction function? The type we assign to prediction functions is a bit complex: the purpose of this section is to explain this complexity layer-by-layer by gradually refining this type.

As a first approximation, a difference between P_1 and P_2 is interpreted as a prediction function of type

$$d \times \delta c \twoheadrightarrow \delta c$$

It takes a prediction direction and an input execution step and produces a predicted execution step. A prediction direction can be either $\downarrow$ to predict from P_1 to P_2, or $\uparrow$ to predict from P_2 to P_1. Notice that a prediction function is partial because the input execution step might not be compatible with the current configuration of the input program and as any program, an oracle can be stuck. The invariants required for an oracle to be sound will prevent such situations.

The previous type gives the raw idea of what a prediction function is. Yet, the reality is more complex than that as we have noticed in the overview of Section 2. First, the executions of the two programs may not be synchronized. Sometimes, it is necessary to wait for the consumption of two or more input execution steps before being able to produce a prediction. This was the case for oracles of assignments swapping: when the oracle is in-between two assignments to be swapped, producing a prediction would not make sense.

Furthermore, when some parts of one reduction chain must be abstracted away, a potentially large number of intermediate execution steps of one of the two programs must be skipped. In the overview section, this was the case when we used the difference language of abstract equivalence: the oracle had to skip all the instructions of the second algorithm by exploiting a proof of its equivalence to the first one. To take care of the desynchronization of some oracles, we made a first refinement step leading to the following type for prediction functions:

$$d \times \delta c \twoheadrightarrow (\mathbb{N} \setminus \{0\} \times \delta c) + \mathbf{wait}$$

With this type, an oracle is asked to produce either a prediction carrying an upper bound on the number of steps it predicts, or the answer **wait**. In the first case, the oracle abstracts away intermediate steps irrelevant to the comparison into a single prediction. In the second case, the oracle acknowledges that it needs to accumulate more input execution steps to produce a meaningful prediction.

This last point makes it necessary to introduce a notion of state in the oracle function which we assumed realized by the values of a type s:

$$s \times d \times \delta c \twoheadrightarrow s \times ((\mathbb{N} \setminus \{0\} \times \delta c) + \mathbf{wait})$$

Being in a purely functional settings, we follow the usual state-passing style: the current state of the oracle is transmitted to the prediction function which produces a new version of this state. To start this process, an oracle must have an initial state. Notice that the oracle state can be arbitrarily rich: any contextual information can influence the prediction. As discussed in the related work, this aspect also raises the expressivity of the framework with respect to the one of prior work like differential symbolic execution or refinement mappings.

Finally, when the oracle is in the middle of a prediction that needs multiple input steps, the direction of prediction cannot be changed. We already witnessed this situation in the overview section when we considered the difference language of assignments swapping.

The prediction is also constrained when one of the program has converged but the other did not: the only way to apply the oracle is to consume a step of

the program that has not converged and to produce **wait** as the prediction for the second program.

The final refinement of the prediction function type introduces this directionality constraint on the next request as a new output of the prediction function:

$$\mathsf{predict}\ \delta c\ s = s \times d \times \delta c \twoheadrightarrow s \times \dot{d} \times ((\mathbb{N} \setminus \{0\} \times \delta c) + \mathbf{wait})$$

where $\dot{d}$ can be $\downarrow$ to only allow the next request to be from P_1 to P_2, $\uparrow$ to only allow the next request to be from P_2 to P_1 and $\downarrow\uparrow$ to allow both directions in the next request. In Section 6, we will explain why this directionality imposed by the internal state of the oracle has important consequences on its composability with other oracles.

Definition 3. *The compatibility relation $\preceq$ between allowed directions is defined as follows (i) $\downarrow \preceq \downarrow\uparrow$; (ii) $\uparrow \preceq \downarrow\uparrow$; (iii) $\forall \dot{d}, \dot{d} \preceq \dot{d}$.*

Dealing with convergence, divergence and crash The prediction function is always terminating and, as seen above, is to be called in an appropriate direction with a reduction step of the program to predict from. It is the responsibility of the caller—which is not required to terminate, thus covering infinite executions—to compute this reduction step and call the the prediction function with the appropriate program states, oracle state, reduction step and compatible direction.

By convention, if one of the two program has converged, the only valid prediction about its reduction is **wait**. In addition, to witness the fact that one of the two programs will be stuck in less than k steps, the prediction is written $(k, \mathbf{stuck})$.

A formal definition for difference languages

Definition 4. *A* difference language *for a language equipped with a change-indexed operational semantics over δc is a tuple $(\Delta, s, i, \mathcal{O})$ such that Δ is a set of terms, s is the type of oracle states, i is the initial oracle state of s and $\mathcal{O}$ is a function of type $\Delta \to \mathsf{predict}\ \delta c\ s$.*

An oracle is a program. Hence, proving properties about this program gives properties about the difference it describes. Proving properties about the function $\mathcal{O}$ of a difference language gives properties about all the differences of this language.

Universal difference language For any language, one can define a universal difference language that relates any pair of programs. Indeed, it suffices to embed one interpreter per program in the prediction function so that there is no prediction at all but simply a standard evaluation to produce the oracle output. An alternate universal oracle language could be achieved by predicting **wait** at each step.

Of course, this difference language is not interesting. First, the design of a difference language is precisely *not to interpret* programs but deduce one execution from the other one with a minimal amount of dynamic information. Second, the existence of a difference between two programs should witness some form of closeness between these two programs: if each program is close to all the others, then this notion of closeness is trivial and not informative.

Identity difference language For any language, it is also possible to define a trivial difference language—the so-called identity language—that relates every program only to itself. The prediction function for this language simply returns the input steps as a prediction.

This difference language is a bit more interesting than the universal difference language since it relates all the programs that produce the same reduction chains whatever their source code is.

In the case of Imp though, the initial configuration stores the entire program in its continuation which implies that only syntactically equal programs will be related by an oracle of the identity difference language.

4.2 Difference languages on Imp

In this section, we sketch several formalizations of simple difference languages that we found interesting to compare programs written in the Imp programming language. These difference languages fall into two categories: the first category consists in difference languages that relate extensionally equivalent programs which differ intentionally ; the second category includes difference languages that relate programs which are observationally distinct but whose difference of behavior can be finitely captured. By lack of space, these formal definitions are only sketched. We encourage the reader to look at the Coq development to get more details about the definitions and the properties of these difference languages.

Equivalences up-to

Renamings The difference language of renamings characterizes programs that are equivalent up to a well-chosen renaming of their variables.

In that case, a difference is fully characterized by a renaming ϕ. Its interpretation is the oracle that simply rewrites with ϕ the variables that appear in the memory change of the input step while reusing the continuation change. Such an oracle is global: it has no state and is independent from the current configurations of programs.

Definition 5. *The difference language* $\Delta^{Ren}_{\text{IMP}}$ *is the tuple* $(\Delta, s, i, \mathcal{O})$ *such that* Δ *is the set of bijective variable mappings* ϕ*, the type of oracle states* s *is* $\mathbf{1}$*, the initial oracle state* i *is* $()$ *and the function* $\mathcal{O}$ *is such that:*

$$\mathcal{O}(\phi)((), d, \delta c) = ((), \downarrow\uparrow, (1, \phi(\delta c)))$$

where ϕ *is extended to configuration changes the obvious way.*

Branch permutation The difference language of branch permutation characterizes programs that are equivalent up to the following equation:

$$\textbf{if } (b)\ C_1 \textbf{ else } C_2 = \textbf{if } (!b)\ C_2 \textbf{ else } C_1$$

Contrary to the previous difference language, a difference in the language of branch permutations witnesses a *local* modification. Hence, the difference is described by a path π in the abstract syntax tree which locates the application of the equation and a condition modifier $\Diamond$ that indicates if the condition must be negated or if the already present negation must be removed.

A path π is a sequence of natural numbers: each natural number corresponds to the subterm to go through to reach the point where the modification takes place.

$$\pi ::= \epsilon \mid 0 \cdot \pi \mid 1 \cdot \pi$$

As the number of children of an Imp syntactic construction never exceeds 2, the case 0 and 1 are enough to reach any subterm of an Imp program from the root of its abstract syntax tree. A condition difference ρ is simply the identity or a condition modifier located at some path of the source code:

$$\rho ::= \mathbf{Id} \mid \Diamond[\pi] \qquad \Diamond ::= \neg \mid \neg^{-1}$$

This path is static. Thus, it must be rewritten by the oracle along the evaluation to be rephrased in terms of the current continuation so that the oracle can dynamically detect where it has reached the modification point. The image of this translation is called a *continuation modifier*. It is a list of differences $\overline{\rho}$ of the same length as the current continuation.

The oracle maintains a continuation modifier in its internal state to determine its behavior with respect to the current continuation. By inspection of the input steps, the continuation modifier is decomposed until it has the shape $\Diamond[\epsilon]\cdot\overline{\rho}$ which means that the modification $\Diamond$ must be applied immediately. Notice that the modification may be done several times during the oracle evaluation since the modification point may be enclosed in a loop. The decomposition rules are realized by the partial function $\Xi(\delta\kappa, \rho\cdot\overline{\rho})$ specified in Figure 3. Roughly speaking, this function mimicks the function $\Delta\mathsf{step}$ except that it pushes the change $\Diamond$ to the subcontinuations specified by the path π. In the case for **while**, the change is duplicated for each iteration of the loop.

The interpretation of a difference $\Diamond$ switches selections of **then**-branches and selections of **else**-branches:

$$\begin{aligned} u\mathbf{else}^{\leftrightarrow} &= u\mathbf{then} \\ u\mathbf{then}^{\leftrightarrow} &= u\mathbf{else} \\ \delta\kappa^{\leftrightarrow} &= \mathbf{undefined} \text{ otherwise} \end{aligned}$$

Formally, this oracle is defined as follows:

$$\mathcal{O}(\Diamond[\pi])(\overline{\rho}, d, (\delta M, \delta\kappa)) = (\overline{\rho}', \lfloor\,\rfloor, (1, \delta c))$$

where

$$(\overline{\rho}', \delta c) = \begin{cases} (\overline{\rho}'', (\delta M, \delta\kappa^{\leftrightarrow})) & \text{if } \overline{\rho} = \Diamond[\epsilon]\cdot\overline{\rho}'' \\ (\Xi(\delta\kappa, \overline{\rho}), \quad (\delta M, \delta\kappa)) & \text{otherwise} \end{cases}$$

$$
\begin{array}{rl|rl}
\Xi(\mathbf{pop}, \rho \cdot \overline{\rho}) &= \overline{\rho} & \Xi(u\mathbf{then}, \mathbf{Id} \cdot \overline{\rho}) &= \mathbf{Id} \cdot \overline{\rho} \\
\Xi(u\mathbf{seq}, \mathbf{Id} \cdot \overline{\rho}) &= \mathbf{Id} \cdot \mathbf{Id} \cdot \overline{\rho} & \Xi(u\mathbf{then}, \Diamond[0 \cdot \pi] \cdot \overline{\rho}) &= \Diamond[\pi] \cdot \overline{\rho} \\
\Xi(u\mathbf{seq}, \Diamond[0 \cdot \pi] \cdot \overline{\rho}) &= \Diamond[\pi] \cdot \mathbf{Id} \cdot \overline{\rho} & \Xi(u\mathbf{then}, \Diamond[1 \cdot \pi] \cdot \overline{\rho}) &= \mathbf{Id} \cdot \overline{\rho} \\
\Xi(u\mathbf{seq}, \Diamond[1 \cdot \pi] \cdot \overline{\rho}) &= \mathbf{Id} \cdot \Diamond[\pi] \cdot \overline{\rho} & \Xi(u\mathbf{else}, \mathbf{Id} \cdot \overline{\rho}) &= \mathbf{Id} \cdot \overline{\rho} \\
\Xi(u\mathbf{while}, \mathbf{Id} \cdot \overline{\rho}) &= \mathbf{Id} \cdot \mathbf{Id} \cdot \overline{\rho} & \Xi(u\mathbf{else}, \Diamond[0 \cdot \pi] \cdot \overline{\rho}) &= \mathbf{Id} \cdot \overline{\rho} \\
\Xi(u\mathbf{while}, \Diamond[1 \cdot \pi] \cdot \overline{\rho}) &= \Diamond[\pi] \cdot \Diamond[1 \cdot \pi] \cdot \overline{\rho} & \Xi(u\mathbf{else}, \Diamond[1 \cdot \pi] \cdot \overline{\rho}) &= \Diamond[\pi] \cdot \overline{\rho}
\end{array}
$$

Fig. 3. Updating a continuation modifier.

Definition 6. *The difference language* $\Delta_{\text{IMP}}^{SwapBranches}$ *is the tuple* $(\Delta, s, i, \mathcal{O})$ *such that* Δ *is the set of localized condition modifications* ρ*, the type of oracle states* s *is* $\overline{\rho}$*, the initial oracle state* i *is* $\Diamond[\pi]$ *and the function* $\mathcal{O}$ *is defined as above.*

Example 1. The $\Delta_{\text{IMP}}^{SwapBranches}$ oracle $\neg[1 \cdot \epsilon]$ describes the difference between the two following programs:

```
1    x = 12;
2    if (x < 0)
3      y = 42;
4    else
5      y = 0;
```

```
1    x = 12;
2    if (! (x < 0))
3      y = 0;
4    else
5      y = 42;
```

Assignment swappings The difference language of assignment swappings characterizes programs that are equivalent up to reordering of successive independent assignments. The location of the assignment reordering is represented by a path π in the abstract syntax tree. Tracking down the place where the modification takes place is implemented using the same mechanism as in the previous section except that the local change $\Diamond$ is now an assignment swapping.

As pointed out in the Section 2, when the oracle is given an assignment of one program P_1 that is swapped with another assignment in the other program P_2, it must retain this assignment in its state until the next assignment of P_1 is provided. Therefore, in addition to the continuation modifier machinery, the oracle must also implement a two-state machine:

- In State "S_0", the oracle looks at the command C at the top of the continuation, if it is an assignment that is swapped in the modified program then it stores this assignment step δc, goes to state "$S_1(\delta c)$", returns **wait** as a prediction ; otherwise if C is any other command, it simply produces the input execution step as a prediction and stays in State S_0.
- In State "$S_1(\delta c)$", the oracle takes the next input step $\delta c'$ of P_1 and returns the composition of $\delta c'$ followed by δc as a prediction for P_2. Then, the oracle goes to State S_0.

When the oracle goes from State S_0 to State "$S_1(\delta c)$" with a **wait** prediction, it also returns a directionality constraint to force the subsequent request to be

done in the same direction (from P_1 to P_2 in our case). On the contrary, when the oracle goes from State "$S_1(\delta c)$" to State S_0 or stays in State S_0, no directionality constraints is imposed.

Definition 7. *The difference language $\Delta_{\text{IMP}}^{SwapAssign}$ is the tuple $(\Delta, s, i, \mathcal{O})$ such that Δ is the set of paths π where the assignment swapping occurs, the type for oracle states s is $\overline{\rho} \times (S_0 + S_1 : \delta c)$, the initial states i is $(\Diamond[\pi], S_0)$ and $\mathcal{O}$ is an oracle that implements the informal description given above.*

Reparenthesized sequences The difference language of sequences reparenthesizing characterizes programs that are equivalent up to the following equation:

$$C_1; (C_2; C_3) = (C_1; C_2); C_3$$

This difference language is similar to the language of assignment swappings in the sense that the two compared programs are locally desynchronized when the input execution step is in the middle of the sequence to be reparenthesized. This time though, the oracle does not generate commands that are late in the modified programs but modifications of the (modified or source) program continuation to align the two programs continuations.

More precisely, in State S_0, the oracle is consuming a path π to reach the point where the equation is to be applied. Then, there are two possible states S_1 and S_2 depending on the orientation of the equation application.

In State S_1, if P_1 is the program with the command of the shape $C_1; (C_2; C_3)$ and if the request to the oracle is to predict P_2 from P_1, then the oracle is given a u**seq** as input and must returns two u**seq** as a prediction to lift C_1 at the top of the continuation in P_2.

In State S_2, if P_1 is the program with the command of the shape $(C_1; C_2); C_3$ and if the request to the oracle is to predict P_2 from P_1, then the oracle is given a first u**seq** as input. It cannot produce anything but **wait** because the next step of P_2 consists in the execution of C_1. The oracle also forbids the prediction direction to change so that when the oracle is given the second u**seq**, the continuations of P_1 and P_2 are aligned. Afterwards, the oracle can behaves like the identity.

Definition 8. *The difference language $\Delta_{\text{IMP}}^{SeqAssoc}$ is the tuple $(\Delta, s, i, \mathcal{O})$ such that Δ is the set of paths π where a sequence is reparenthesized. the type of oracle states s is $\overline{\rho} \times (S_0 + S_1 + S_2)$, the initial oracle state i is $(\Diamond[\pi], S_0)$ and $\mathcal{O}$ is an oracle that implements the informal description given above.*

Abstract equivalence Given two programs with distinct commands C_1 and C_2 at path π but otherwise identical, if there exists a proof Π that the commands C_1 and C_2 are functionally equivalent, then the difference language of abstract equivalences has a difference whose interpretation witnesses this fact.

As long as the path π is not reached by the oracle in the reduction chains of the two compared programs, the oracle behaves as the identity. When the path π is reached, if an execution step of P_1 is provided as input, the oracle starts consuming all the execution steps corresponding to the command C_1 of

P_1 and produces the prediction **wait** and the allowed direction $\downarrow$ (so that the oracle is only called in this direction until the command C_1 is entirely executed).

After the execution of C_1, the change δc performed by the command C_1 is known to the oracle and it can use it to produce a prediction for P_2 that simply pops the command C_2 from the top of P_2's continuation and applies δc on the resulting configuration.

A problem remains since a prediction must provide an upper bound on the number of steps needed to execute δc on the predicted program. How to compute such an upper bound? There are several answers to this question. One possibility is to assume that the proof of functional equivalence comes with constructive termination proofs for the two commands C_1 and C_2. As we are in Type Theory, concrete bounds on the number of execution steps can be extracted from these proofs. Actually, there is another possibility: we can do without any termination proofs by only allowing predictions from P_2 to P_1, so that the caller is forced to execute C_2. If the execution of C_2 does not converge, then the oracle will produce an infinite sequence of **wait**, which testifies that nothing can be said about how the final configurations of the two programs can be compared.

In our Coq development, we offer the two possibilities as two distinct difference languages $\Delta_{\text{IMP}}^{AbstractEquiv}$ and $\Delta_{\text{IMP}}^{AbstractEquivNoBound}$. Of course, a difference in $\Delta_{\text{IMP}}^{AbstractEquivNoBound}$ is a weaker result than a difference in $\Delta_{\text{IMP}}^{AbstractEquiv}$.

The type of the oracle state in $\Delta_{\text{IMP}}^{AbstractEquiv}$ is

$$s = S_0 : \overline{\rho} + S_1 : d \times \mathbb{N} \times \overline{\rho} \times \delta M$$

In the state $S_0(\overline{\rho})$, the oracle consumes the input steps until it reaches C_1 and C_2, then it enters state $S_1(d, n, \overline{\rho}, \epsilon)$ where d is the direction of the request that reached the modification point, n is the depth of the continuation representing the execution of C_1, $\overline{\rho}$ is the remaining continuation modifiers and ϵ is the empty configuration change. Once in the state $S_1(d, n, \overline{\rho}, \delta M)$, the oracle forces the direction d for further requests until it has $n = 0$, that is to say until the command C_1 is entirely executed. During these silent predictions, the oracle accumulates the memory changes in δM. When $n = 0$, the oracle produces $(m, (\mathbf{pop}, \delta M))$ as a prediction and goes back to state $S_0(\overline{\rho})$. The natural number m is an upper bound on the number of steps needed for the convergence of C_2 : we assume that it is provided by the proof accompanying the difference.

Definition 9. *The difference language $\Delta_{\text{IMP}}^{AbstractEquiv}$ is the tuple $(\Delta, s, i, \mathcal{O})$ such that Δ is the set of pairs made of a path π, a proof of equivalences between the commands at this path and two proofs of termination for these commands. The type of oracle states s is defined as above. The initial oracle state i is $S_0(\Diamond[\pi])$ and $\mathcal{O}$ is an oracle that implements the informal description given above.*

The type of the oracle state in $\Delta_{\text{IMP}}^{AbstractEquivNoBound}$ is

$$s = S_0 : \overline{\rho} + S_1 : d \times \mathbb{N} \times \overline{\rho} \times \delta M + S_2 : d \times \mathbb{N} \times \overline{\rho}$$

In addition to the two states already present in the oracles of $\Delta_{\text{IMP}}^{AbstractEquiv}$, the extra state $S_2(d, m, \overline{\rho})$ is activated when the oracle waits for the convergence of the execution of C_2. As long as the execution of C_2 is not finished, the oracle maintains the depth m of the part of the continuation containing the remaining commands of C_2. When $m = 0$, the command C_2 has converged and the oracle goes back to state $S_0(\overline{\rho})$ producing the same prediction as in $\Delta_{\text{IMP}}^{AbstractEquiv}$.

Definition 10. *The difference language* $\Delta_{\text{IMP}}^{AbstractEquivNoBound}$ *is the tuple* $(\Delta, s, i, \mathcal{O})$ *such that* Δ *is the set of pairs made of a path* π *and a proof of equivalences between the commands at this path. The type of oracle states* s *is defined as above. The initial oracle state* i *is* $S_0(\Diamond[\pi])$ *and* $\mathcal{O}$ *is an oracle that implements the informal description given above.*

Characterized inequivalences

Crash avoidance A large class of programming errors are induced by not respecting preconditions. Typically, the following instruction

```
1  z = x / y;
```

assumes that `y` is not equal to zero. If the preceding instructions do not satisfy this condition, the program will crash. To solve that class of bug, the programmer can defensively introduce an **if**-statement:

```
1  if (not (y = 0)) {
2    ...
3    z = x / y;
4  } else
5    ...
```

which avoid the behavior of the original program that crashes in the **then**-branch. Notice that the instructions in the **else**-branch may also crash but in a different configuration.

The difference language of crash avoidance characterizes pair of programs related by such a bug fix. The oracles of this language behaves as the identity until the location of the bug fix is reached by the evaluation. Then, once the location is reached, the countdown begins: the buggy program will crash soon. More precisely, the difference comes with a proof that the crash will happen in less than n steps for some n. In this state, if the oracle is asked to predict the crashing program's behavior, it will predict a delta of **stuck** with an upper bound of n steps.

In this difference language, the type of oracle state is

$$s = S_0 : \overline{\rho} + S_1 : \mathbb{N} + S_2$$

In the state $S_0(\overline{\rho})$, the oracle is waiting for the point where the bug fix is applied. Once it has reach this point, the oracle enters $S_1(n)$ decrementing n each time the caller provides an execution step of the buggy program as input. In less than n of such input steps or predictions of execution steps of the buggy program, the failure leads the oracle to the final state S_2 in which only a prediction of **stuck** can be issued.

Definition 11. *The difference language* $\Delta^{CrashFix}_{\text{IMP}}$ *is the tuple* $(\Delta, s, i, \mathcal{O})$ *such that* Δ *is the set of triples made of a path* π*, a tuple* (b, C) *to describe the* ***if*** *statement implementing a fix, and a proof of that one of the programs will be stuck at most* n *steps after reaching path* π *if the condition* b *is not met. The type of oracle states* s *is defined as above. The initial oracle state* i *is* $S_0(\Diamond[\pi])$ *and* $\mathcal{O}$ *is an oracle that implements the informal description given above.*

Example 2. The $\Delta^{CrashFix}_{\text{IMP}}$ oracle $(1 \cdot \epsilon, (y \neq 1, y = 0; sum = 42), H)$ where H is a proof of a crash within 4 steps, describes the difference between the two following programs:

```
1      x = x + 2;
2
3      y = y - 1;
4      count = x / y;
5      sum = x * count;
6
7
8
9
```

```
1      x = x + 2;
2      if (y != 1) {
3        y = y - 1;
4        count = x / y;
5        sum = x * count;
6      } else {
7        y = 0;
8        sum = 42;
9      }
```

Distinct output values The previous difference language of crash avoidance characterizes some form of temporal difference between two programs: before the crash, the two programs behave similarly; after the crash, only one program can be reduced.

It is also possible to characterize pairs of programs that have reduction chains of same length but holding different configurations. Indeed, consider two programs with the same control-flow but different assignments: these two programs will compute distinct output values but in somewhat "the same way".

The difference language of distinct output values assume that variables are split into two different categories: the input variables and the output variables. The input variables influence the control-flow whereas the output variables do not. Two programs are related by this difference language if they share the same classifications of their variables and the same boolean expressions on **if**-statements and **while**-statements. As said earlier, the assignments of the output variables can be different in each program (provided that two assignments at the same path either get stuck together or executes without error).

More precisely, a difference in this language is a pair formed by a path to an assignment that differs in the two programs and a list of variables, namely the output variables, that are impacted by this change.

Once the oracle has reached the path where the assignment modification occurs the oracle maintains two stores in parallel, one for each program. Given an input execution step δc which is not an assignment, the oracle behaves as the identity. Otherwise, it executes the two distinct assignments separately in their dedicated store.

In this difference language, the type of oracle state is

$$s = S_0 : \overline{\rho} + S_1 : M \times M$$

In the state $S_0(\overline{\rho})$, the oracle is waiting for the point where the two programs differ. Once there, it moves to the state $S_1(M, M)$ and it assigns output variables in the two distinct stores depending on the considered program.

Definition 12. *The difference language* $\Delta_{\text{IMP}}^{ValueChange}$ *is the tuple* $(\Delta, s, i, \mathcal{O})$ *such that* Δ *is the set of pairs made of a path* π *and a list of output variables. The type of oracle states* s *is defined as above. The initial oracle state* i *is* $S_0(\Diamond[\pi])$ *and* $\mathcal{O}$ *is an oracle that implements the informal description given above.*

Example 3. The $\Delta_{\text{IMP}}^{ValueChange}$ oracle $(0 \cdot \epsilon, [x; pow])$ describes the difference between the two following programs:

```
x = 42;
count = 5;
pow = 1;
while (0 < count) {
  count = count - 1;
  pow = pow * x;
}
```

```
x = 10;
count = 5;
pow = 1;
while (0 < count) {
  count = count - 1;
  pow = pow * x;
}
```

Abstract inequivalence Much like with $\Delta_{\text{IMP}}^{AbstractEquiv}$, it is possible to abstract from the small-step semantics presentation to reason about two semantically different commands C_1 and C_2 of two otherwise identical programs. The difference language of abstract inequivalences describes such changes, given a proven bijection *translate* between the stores resulting from an execution of C_1 and C_2.

Just like $\Delta_{\text{IMP}}^{AbstractEquiv}$, the oracle behaves as the identity as long as the path π is not reached. When the path π is reached, if an execution step of P_1 is provided as input, the oracle starts consuming all the execution steps corresponding to the command C_1 of P_1 and produces the prediction **wait** and the allowed direction $\lfloor$.

When C_1 has been completely executed, the oracle uses the given bijection to compute the result of executing C_2, and produce a prediction for P_2 that simply pops the command C_2 and applies the computed changes on stores. After this, the oracle does not know how to relate the programs further, and will simply return **wait**.

The type of the oracle state in $\Delta_{\text{IMP}}^{AbstractInequiv}$ is

$$s = S_0 : \overline{\rho} + S_1 : d \times \mathbb{N} \times \overline{\rho} \times \delta M + S_2$$

In the state $S_0(\overline{\rho})$, the oracle consumes the input steps until it reaches C_1 and C_2, then it enters state $S_1(d, n, \overline{\rho}, \epsilon)$ where d is the direction of the request that reached the modification point, n is the depth of the continuation representing the execution of C_1, $\overline{\rho}$ is the remaining continuation modifiers and ϵ is the empty configuration change. Once in the state $S_1(d, n, \overline{\rho}, \delta M)$, the oracle forces the direction d for further requests until it has $n = 0$, that is to say until the command C_1 is entirely executed. During these silent predictions, the oracle accumulates the memory changes in δM. When $n = 0$, the oracle produces

$(m, (\mathbf{pop}, \delta M'))$ as a prediction, where $\delta M'$ is a change on stores such that $M \oplus \delta M' = translate(M \oplus \delta M)$, and the natural number m is an upper bound on the number of steps needed for the convergence of C_2 : we assume that it is provided by the proof accompagnying the difference. The oracle then switches to state S_2 in which it returns **wait** no matter the input.

Definition 13. *The difference language* $\Delta_{\text{IMP}}^{AbstractInequiv}$ *is the tuple* $(\Delta, s, i, \mathcal{O})$ *such that* Δ *is the set of pairs made of a path* π*, a proven bijection between the stores resulting from execution of the commands at this path, and proofs of termination for these commands. The type of oracle states* s *is defined as above. The initial oracle state* i *is* $S_0(\Diamond[\pi])$ *and* $\mathcal{O}$ *is an oracle that implements the informal description given above.*

Example 4. The difference between the following two programs can be described by an oracle of $\Delta_{\text{IMP}}^{AbstractInequiv}$ relating the sub-programs at path $1 \cdot 1 \cdot \epsilon$ using

$$translate\, M_0\, M = \begin{cases} M & \text{if } M_0[count] < 1 \\ M_0' & \text{if } M_0[count] < 2 \\ M_0'[sum := M[x]] & \text{otherwise} \end{cases}$$

where $M_0' = M_0[count := 1, y := M[x], x := M[y] - M[x]]$:

```
1   x = 1;
2   y = 1;
3   // Inequivalent part:
4   x = x;
5   y = y;
6   while (0 < count) {
7     sum = x + y;
8     x = y;
9     y = sum;
10    count = count - 1;
11  }
```

```
1   x = 1;
2   y = 1;
3   // Inequivalent part:
4   x = x;
5   y = y;
6   while (1 < count) {
7     sum = x + y;
8     x = y;
9     y = sum;
10    count = count - 1;
11  }
```

5 Properties over difference languages

5.1 Soundness

Given a difference $\delta \in \Delta$ and two programs P_1 and P_2, we formally define in this section under which conditions the difference δ is a sound difference for P_1 and P_2.

Configurations correlated by an oracle As coined in the introduction, the behavior of an oracle is characterized by all the possible sequences of directed predictions it can perform. Each prediction is either performed from a known configuration c_1 of P_1 to a configuration c_2 of P_2 or from a known configuration c_2 of P_2 to a configuration c_1 of P_1. Besides, as we have seen earlier, at some point, the oracle may force a specific direction for the next prediction. The pairs

$$\textsc{C-Start}\quad \frac{}{\mathcal{I}(P_1) \sim_\delta \mathcal{I}(P_2) \Downarrow (i, \downarrow\uparrow)}$$

$$\textsc{C-StepFromLeft}\quad \frac{c_1 \sim_\delta c_2 \Downarrow (s_2, \dot{d}_1) \qquad \downarrow \preceq \dot{d}_1 \qquad \mathcal{O}_\delta(s_2, \downarrow, \Delta\mathsf{step}(c_1)) = (s_3, \dot{d}_2, (n, \delta c_2))}{c_1 \oplus \Delta\mathsf{step}(c_1) \sim_\delta c_2 \oplus \delta c_2 \Downarrow (s_3, \dot{d}_2)}$$

$$\textsc{C-StepFromRight}\quad \frac{c_1 \sim_\delta c_2 \Downarrow (s_2, \dot{d}_1) \qquad \uparrow \preceq \dot{d}_1 \qquad \mathcal{O}_\delta(s_2, \uparrow, \Delta\mathsf{step}(c_2)) = (s_3, \dot{d}_2, (n, \delta c_1))}{c_1 \oplus \delta c_1 \sim_\delta c_2 \oplus \Delta\mathsf{step}(c_2) \Downarrow (s_3, \dot{d}_2)}$$

$$\textsc{C-StopRight}\quad \frac{c_1 \sim_\delta c_2 \Downarrow (s_2, \dot{d}_1) \qquad \downarrow \preceq \dot{d}_1 \qquad \mathcal{O}_\delta(s_2, \downarrow, \Delta\mathsf{step}(c_1)) = (s_3, \dot{d}_2, \bot)}{c_1 \oplus \Delta\mathsf{step}(c_1) \sim_\delta c_2 \Downarrow (s_3, \dot{d}_2)}$$

$$\textsc{C-StopLeft}\quad \frac{c_1 \sim_\delta c_2 \Downarrow (s_2, \dot{d}_1) \qquad \uparrow \preceq \dot{d}_1 \qquad \mathcal{O}_\delta(s_2, \uparrow, \Delta\mathsf{step}(c_2)) = (s_3, \dot{d}_2, \bot)}{c_1 \sim_\delta c_2 \oplus \Delta\mathsf{step}(c_2) \Downarrow (s_3, \dot{d}_2)}$$

Fig. 4. Configurations correlated by a difference.

of configurations related by the oracle through its (potentially non terminating) interaction with the caller are called the configurations correlated by the oracle. The set of these relations is the informative content of each oracle.

More formally, we say that "The two configurations c_1 of P_1 and c_2 of P_2 are correlated by the oracle of δ and for the next prediction, the allowed directions are $\dot{d}$ and the oracle state is s_2" if the judgment $c_1 \sim_\delta c_2 \Downarrow (s_2, \dot{d})$ can be derivated from the rules of Figure 4.

The rule C-Start asserts that any sequence of correlated configurations starts with two initial configurations and with an oracle whose state is also initial. The rule C-StepFromLeft states that from two configurations c_1 and c_2 whose correlation has been established by an oracle ending at state s_2 and allowing $\dot{d}_1$. If $\dot{d}_1$ is compatible with $\downarrow$ then the oracle can be called in that direction with the state s_2 and with the input step δc_1 of P_1 to predict n reduction steps of P_2 represented by δc_2. It suffices to apply the two configuration changes to get the next correlated states. Besides, in addition to the prediction, the oracle has returned the new state s_3 as well as the allowed direction $\dot{d}_2$ for the next prediction. The rule C-StepFromRight is similar to the rule C-StepFromLeft except that the prediction direction is $\uparrow$ and hence the roles of δc_1 and δc_2 are exchanged. The rules C-StopRight and C-StopLeft account for the case where the prediction has made no progress on the reduction chains targeted by the request to the oracle. This can happen if the oracle needs more input steps to make a decision about the prediction or because the reduction chains is ended, either because the program has converged or because it is stuck.

Validation of predictions A prediction of the form **wait** is always valid. On the contrary, the other forms of predictions must always correspond to actual future configurations to be valid.

Definition 14 (Valid predictions of execution). *A prediction* $(\dot{d}, s, (n, \delta c))$ *is valid for a configuration* c *if the natural number* n *is greater or equal to the strictly positive number of reduction steps needed to reach the configuration* $c \oplus \delta c$ *from* c.

Definition 15 (Valid predictions of failure). *A prediction* $(\dot{d}, s, (n, \textbf{\textit{stuck}}))$ *is valid for a configuration* c *if the natural number* n *is greater than the strictly positive number of reduction steps needed to reach a stuck configuration from* c.

A valid prediction for a configuration c can be concretized as an actual chain of reductions starting from c using the following concretization function:

$$\mathcal{R}^c_{\textbf{stuck}}(c) = c \qquad \mathcal{R}^c_{\delta c}(c_i) = \begin{cases} c_i & \text{if } c_i = c \oplus \delta c \\ c_i \to \mathcal{R}^c_{\delta c}(c_i \oplus \Delta\mathsf{step}(c_i)) & \text{otherwise} \end{cases}$$

Lemma 1. *If* $(\dot{d}, s, (n, p))$ *is a valid prediction for the configuration* c *then the execution of the concretization function* $\mathcal{R}^c_p(c)$ *terminates in less than* n *steps.*

Sound differences To prove that a difference δ is a sound difference between two programs P_1 and P_2, it suffices to show that the predicted configurations appearing in a sequence of correlated configurations are actual configurations of the programs reduction chains. We formulate this property as a local requirement on each step of the correlation sequence.

Definition 16. *A difference* δ *is sound for* P_1 *and* P_2 *if* $c_1 \sim_\delta c_2 \Downarrow (s_2, \dot{d})$ *implies that (i) if* $\lfloor \preceq \dot{d}$, *then* $\mathcal{O}_\delta(s_2, \lfloor, \Delta\textit{step}(c_1))$ *is a valid prediction and (ii) if* $\lceil \preceq \dot{d}$, *then* $\mathcal{O}_\delta(s_2, \lceil, \Delta\textit{step}(c_2))$ *is a valid prediction.*

Notice that this definition prevents $\mathcal{O}(\delta)$ to be undefined on the states and configurations reachable from a pair of correlated states.

Definition 17. *A difference language is sound (resp. canonically sound) if for all difference* δ *there exists a (resp. unique) pair of programs* P_1 *and* P_2 *such that* δ *is sound for* P_1 *and* P_2.

Decidability properties A language of differences can *a priori* characterize any pointwise relation between the configurations of any pairs of reduction chains. This high expressivity may jeopardize the practicability of the approach. We are therefore interested in classes of difference languages with decidability results.

Definition 18. *A language of differences has a* decidable checking problem *if there exists an algorithm that decides for any triple* (δ, P_1, P_2) *if* δ *is a sound difference between* P_1 *and* P_2.

Definition 19. *A language of differences has a* decidable inference problem *if there exists a sound and complete algorithm that computes for any pair of programs* (P_1, P_2) *a sound difference* δ *between* P_1 *and* P_2 *if such a difference exists in that language.*

Applicative difference languages Some languages of differences are inherently static because they capture a source-to-source transformation.

Definition 20. *A language of differences is applicative if there exists a function $\mathcal{A}$ of type $P \times \delta \to P$ such that for any sound difference δ between P_1 and P_2, we have $\mathcal{A}(P_1, \delta) = P_2$.*

Bijective difference languages Because of the directionality constraints that may be imposed by an oracle, the existence of a difference δ between P_1 and P_2 might not imply the existence of a difference between P_2 and P_1.

Definition 21. *A language of differences is bijective if there exists a function $\bullet^{-1}$ of type $\delta \to \delta$ such that for any sound difference δ between P_1 and P_2, δ^{-1} is a sound difference between P_2 and P_1.*

5.2 About nondeterminism

The framework of differential operational semantics can probably be extended to handle nondeterministic languages. Roughly speaking, oracles could have lived in the nondeterministic monad, i.e. by predicting all possible execution steps out of all possible input steps compatible with all possible configurations reachable after every valid sequence of requests. In that case, the interpretation of a difference would not be a single relation between the configurations of the two reduction chains of the compared programs but instead a set of relations between the configurations taken in two sets of reduction chains of the compared programs. In our opinion, the combinatorial explosion induced in that context would have made the mechanical proofs unfeasible and, what is more important, oracles would have been too difficult to grasp by programmers.

6 Composition of differences

6.1 Composition of oracles

Given a sound difference δ_1 between P_1 and P_2 written in a language of differences Δ_1 and another sound difference δ_2 between P_2 and P_3 written in potentially distinct language of differences Δ_2, how can a sound difference $\delta \in \Delta$ between P_1 and P_3 be constructed by using P_2 as an intermediate program and δ as the composition of δ_1 and δ_2? We define the composition of two differences δ_1 and δ_2 as an oracle that embeds the state of the two underlying oracles and whose prediction function calls one of the underlying oracles' prediction function, and then runs the other oracles' on each step of the concretization of the result, returning the last prediction along with the largest allowed direction compatible with both oracles. Unfortunately, the aforementionned prediction function is not always defined. Indeed, if the first underlying oracle predicts a crash of P_2, this prediction cannot be concretized for use with the second oracle. Furthermore, since the second oracle's prediction function is called repeatedly with the same

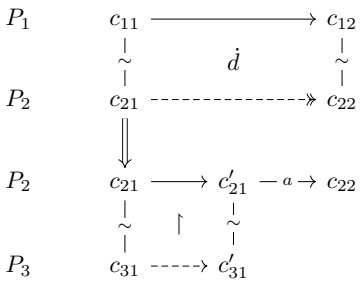
direction on all concretized steps of the first oracle's prediction, it may be undefined as a result of it imposing an incompatible direction. We provide two sufficient conditions on oracles to prevent those issues and guarantee that their compositions are sound (Section 6.2).

Assume that the oracle of a difference δ_1 is a partial function of type $\mathsf{predict}\ \delta c\ s_1$ and the oracle of a difference δ_2 is a partial function of type $\mathsf{predict}\ \delta c\ s_2$. From these two types, we can formally construct the type $\mathsf{predict}\ \delta c\ (s_1 \times s_2)$. In other words, we can implement an oracle that maintains the states of the two oracles $\mathcal{O}(\delta_1)$ and $\mathcal{O}(\delta_2)$ side-by-side. Now, how would it relate the configurations of P_1 and P_3 using $\mathcal{O}(\delta_1)$ and $\mathcal{O}(\delta_2)$?

Let us assume that the composition oracle is used in the direction $\downarrow$. The composition oracle will apply the oracle $\mathcal{O}(\delta_1)$ on the input execution step and concretize its prediction into a sequence of input execution steps for the second oracle $\mathcal{O}(\delta_2)$: the composition of the predictions made by $\mathcal{O}(\delta_2)$ gives the prediction of the composition oracle. In the direction $\uparrow$, the roles of $\mathcal{O}(\delta_1)$ and $\mathcal{O}(\delta_2)$ are exchanged.

This way of composing differences imposes an obvious restriction on the differences that can be composed: the intermediate program P_2 cannot be stuck as a result of a prediction from one of the two other programs. Indeed, no input step can be concretized from the prediction **stuck** and thus, a request to the second oracle would not be possible. Hence, the oracle of such a composition would end up stuck itself. Anyhow, this restriction makes sense: it would be surprising if a stuck program were an appropriate intermediate point to compare two programs that run safely.

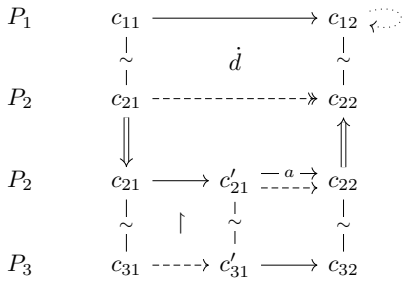
Unfortunately, this definition of composition has a more serious problem. When the second oracle $\mathcal{O}(\delta_2)$ is processing the sequence of execution steps coming from the prediction of $\mathcal{O}(\delta_1)$, what would happen if it forces to reverse the prediction direction for subsequent requests? In that situation, the composition oracle could produce its prediction out of the predictions already produced by $\mathcal{O}(\delta_2)$ but what should be done with the predictions of $\mathcal{O}(\delta_1)$ untouched by $\mathcal{O}(\delta_2)$? The following diagram illustrates this situation:



In this diagram (and the next ones), if the allowed direction for the next prediction is needed by the explanations, it is written in the center of the prediction square. The double arrow represents the operation of concretization of the prediction made by $\mathcal{O}(\delta_1)$ starting at c_{21}. (The double arrow is technically not

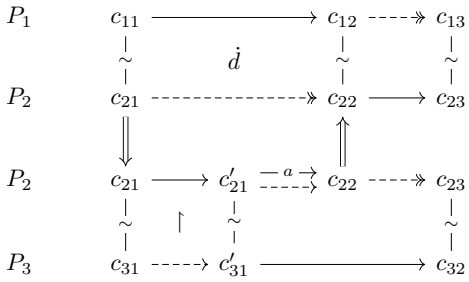
between the two states but between the two arrows. Yet, such notation would complicate the diagram). In this example, just after its first call with the execution step from c_{21} to c'_{21}, the oracle $\mathcal{O}(\delta_1)$ only allows the next requests to predict reduction steps of P_2 from input steps of P_3. Therefore, the transition a from c'_{21} to c_{22} cannot be processed.

A natural—but, as we will see, unsatisfactory—answer to this problem consists in extending the state of the composition oracle to remember the unprocessed steps of the intermediate program and to process them in the subsequent executions of the composition oracle. Indeed, if the composition oracle is called in the other direction (which is $\restriction$ in our example) then the unprocessed steps will coincide with the predicted steps of $\mathcal{O}(\delta_2)$. The unprocessed steps stored in the internal state of the composition oracle will be consumed until new execution steps of P_2 are produced by $\mathcal{O}(\delta_2)$ to resynchronize the reduction of the intermediate program P_2 with the reductions of two external programs P_1 and P_3. During that resynchronization, the oracle of the composition will produce **wait** as long as there are unprocessed steps remaining in the internal state of the composition. This situation is depicted by the following diagram:

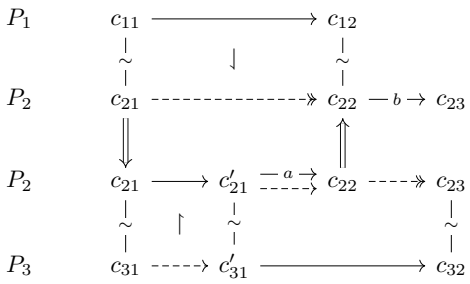


In this diagram, the transition a corresponds exactly to the prediction made by the second oracle. The first oracle is not called at all because no new execution step have been applied to P_2. Thus, no new prediction is produced about P_1 which forces the oracle of the composition to produce a **wait** (depicted using the dotted loop on c_{12}).

If the unprocessed steps are strictly included in the concretization of the prediction made by the second oracle, then the extra execution steps of P_2 are provided as input to the first oracle to obtain a final prediction for P_1:



Now, let us imagine that, in the middle of the resynchronization process described above, the first oracle $\mathcal{O}(\delta_1)$ forces the next prediction to go the other way around. The second oracle $\mathcal{O}(\delta_2)$ can refuse to be used in the direction requested by $\mathcal{O}(\delta_1)$. As a consequence, in spite of the resynchronization mechanism, the oracle of the composition will be stuck again!



In the situation depicted by this diagram, the two oracles choose to only allow directions that are opposite to each other. Hence, when called in both directions, none of the two oracles will accept the concretized predictions of the intermediate program: the composition oracle is stuck.

Therefore, even though $\mathcal{O}(\delta_1)$ and $\mathcal{O}(\delta_2)$ are sound, an incompatibility between their allowed directions annihilates the soundness of their composition.

6.2 Preserving soundness through composition

We chose to restrict ourselves to two properties of oracles that suffice to get the preservation of the soundness by the composition. Our solution is not the final answer to this problem but our restrictions were valid on the difference languages we considered and tractable from the point of view of proof mechanization.

Cooperative oracles To avoid the case where the directions allowed by the two composed oracles are opposite, one restriction is to forbid oracles that force a reversal of prediction direction. Such oracles are called cooperative.

Definition 22. *An oracle $\mathcal{O}(\delta)$ is* cooperative *if for all input direction d, the direction $\dot{d}$ allowed by the oracle for the next prediction is compatible with d, i.e. if $\mathcal{O}(\delta)(s, d, \delta c) = (s', \dot{d}, p)$ then $d \preceq \dot{d}$.*

Theorem 1. *If (i) δ_1 is sound for P_1 and P_2, (ii) δ_2 is sound for P_2 and P_3; and (iii) $\mathcal{O}(\delta_1)$ and $\mathcal{O}(\delta_2)$ are cooperative and (iv) P_2 never gets stuck, then their composition $\mathcal{O}(\delta_1) \cdot \mathcal{O}(\delta_2)$ is sound and cooperative.*

One-step oracle Remember that resynchronization happens when the second oracle is stopped in the middle of the processing of the prediction concretization produced by the first oracle. If the length of this sequence contains at most one step, there is no middle to get stuck in.

	Refl	SeqAssoc	SwapAssign	SwapBranch	AbstractEquiv	AbsEqNoBound	CrashFix	ValueChange	AbstractInequiv
Decidable Checking	✓	✓	✓	✓	✓[1]	✓[1]	✓[1]	✓	✓[1]
Applicative	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cooperative	✓	✓	✓	✓	✓	×	✓	✓	✓
One-step & Bidirectional	✓	×	×	✓	×	×	×	✓	×

Fig. 5. Properties of difference languages on Imp.

Definition 23. *An oracle $\mathcal{O}(\delta)$ is a* one-step oracle *if the concretization of its predictions consists in at most one reduction step.*

Any composition with a one-step oracle preserves productivity.

Theorem 2. *If (i) δ_1 is sound for P_1 and P_2, (ii) δ_2 is sound for P_2 and P_3; (iii) $\mathcal{O}(\delta_1)$ is one-step and always bidirectional; and (iv) P_2 never gets stuck, then their compositions $\mathcal{O}(\delta_1) \cdot \mathcal{O}(\delta_2)$ and $\mathcal{O}(\delta_2) \cdot \mathcal{O}(\delta_1)$ are sound.*

6.3 Synchronized composition

The two conditions described in the previous sections remove the need for an internal state that remembers unprocessed execution steps of the intermediate program. It is therefore possible to define a restricted form of composition called *synchronized composition* that is only defined on oracles that are cooperative or one-step. In our Coq development, we actually focused on this simpler form of composition because it was adapted to almost all our difference languages and it was easier to prove properties about this composition.

Besides, from the programmer perspective, the synchronized composition has an easier interpretation than the more general composition we proposed earlier. Indeed, in the case of the general composition, because of the internal state of the composed oracle, the states of the composed programs can be arbitrarily far from the state of the intermediate program. As a consequence, the programmer must explicit the intermediate program to interpret what the composed oracle does: the programmer must consider each oracle separately to understand which configurations are correlated. On the contrary, in the case of a synchronized composition, the correlation relation between the states of the composed programs can be understood as a composition of the relations of the correlation relation induced by the two oracles. Therefore, the programmer can directly correlate the states of the composed programs without thinking of the states of the intermediate program.

The Figure 5 synthesizes the properties of our difference languages on Imp that have been proved in Coq (✓), or that are known to be unsatisfied (×).

[1] Soundness checking for those oracles is only decidable given underlying proofs

7 Semantic `diff` for Imp programs

Even though automatically inferring program differences given a set of difference languages is obviously undecidable in general (e.g. our abstract equivalence language that would require deciding program equivalences), there are cases where automatically computing program differences is achievable.

In particular, it is trivial for atomic differences expressed in most of the difference languages we presented in this paper (e.g. $\Delta_{\text{IMP}}^{SeqAssoc}$, ...). Since our framework provides a way to compose atomic changes, it is possible to monitor changes to a codebase in real-time and detect atomic differences one after the other, provided the code is indeed edited in an atomic fashion.

This last requirement is, however, largely unrealistic. Therefore, decomposing bigger changes into atomic differences expressed in potentially different languages is also a desirable operation. Unfortunately, the search space grows exponentially with the number of atomic changes considered, and an exhaustive search is not realistic.

While devising algorithms to efficiently decompose bigger changes to atomic differences is an entire line of research that we have barely touched, we wrote a proof-of-concept program that tries to decompose changes into a composition of differences written in $\Delta_{\text{IMP}}^{Ren}$, $\Delta_{\text{IMP}}^{SwapAssign}$, $\Delta_{\text{IMP}}^{SwapBranches}$, and $\Delta_{\text{IMP}}^{ValueChange}$ languages. This program makes use of many different heuristics—mainly based on structured syntactic differences [9]—to find sequences of oracles and associated intermediate programs, and is neither complete nor particularly efficient.

Actually, unlike the framework itself, this prototype is written in OCaml and is not proven sound. However, and more importantly, it uses language definitions extracted from Coq and any found decomposition is checked for soundness using a validator extracted from Coq. Therefore, if an answer is issued by the tools, it can be trusted.

8 Related Work

Equivalences of programs Program equivalence is probably one of the most studied topic in programming language theory. Even the notion of program equivalence itself is subject to many variations depending on what is observed about their evaluation: definitional [14], observational [19], intentional [2], experimental [10], bisimulation-based [15] equivalences and many more have been intensively studied. As illustrated by our examples of difference languages, the general notion of difference languages also captures some equivalence languages. However, even if bisimulations are naturally represented by such an equivalence language, it is not obvious how to define a language for contextual equivalences.

Comparison of inequivalent programs Few general frameworks deal with program comparison. Amongst them, refinement mappings [1] and relational Hoare logic [3] seem to capture the (strict) subset of difference languages whose predictions skip no instruction. They are too concrete to relate two different sorting

algorithms for instance. Formally proving that differential operational semantics is strictly more expressive than these existing systems is for future work. Mutual summaries [11] abstract away procedure implementations leading to more abstract comparison of procedures. Nonetheless, oracles can dynamically choose a specific comparison at each call site while mutual summaries are not aware of call sites.

Quantitative vs qualitative comparison of behaviors Behaviors of distinct programs are measured by means of metric spaces [8], probabilistic bisimulations [12], depth of Böhm trees, or probabilistic tests [13]. Our approach is different from these quantitative approaches since we are looking for qualitative comparisons of program behaviors as syntactic differences, better-suited to programmers.

Semantic patches and refactoring tools Coccinelle's semantics patches [5] are source code transformations specified in a language called SmPL and designed to perform collateral evolutions in system code. As source code transformations, these patches work at the syntactic level but they try to abstract away as many details as possible to augment their applicability. To that end, the process that matches the source code with the patch takes the program control-flow into account thanks to temporal logic formulas. Even if we share similar motivations, our approach is more general because we can model differences between programs that cannot be expressed by static source code transformations as the oracle can exploit the dynamic information stored in its internal state to generate predictions that are context-dependent.

There have been many other attempts to formalize patches as found in control version systems [16]. However, as far as we know, none of them takes the operational semantics of the programming language into account.

The implementation of refactoring tools is an active research topic in software engineering. Except for a simple renaming refactoring tool based on CompCert [7], none of these tools is mechanically certified and unfortunately, none is exempt of bugs as shown by a recent study [25].

Differential static analysis Differential static analysis is an emergent topic in program verification [26, 24]. Differential static analysers typically ensure the preservation of some properties through the evolution of software [23, 22] or try to infer relations between two close versions of the same program [17, 9, 18].

In this work, we do not focus on the problems of inferring or checking differences between programs but on more foundational aspects that will hopefully make it possible to mechanize the proofs of differential static analyzers in the future or to serve as a language for certificates produced by such tools. The line of work about *Differential Symbolic Execution* [18] is probably the approach to comparison of program behaviors which is the closest to ours since these studies are looking for formal characterizations of program differences generated by summaries-directed symbolic interpreters. DSE exploits functional deltas and partition-effects deltas: from our perspective, these deltas are difference written in a low-level difference language expressed by execution paths and variable

mappings. Other difference languages may help summarize the same amount of information and may also turn low-level differences into higher-level ones. Consider:

```
P1: if (x > 0) return 1; else if (x < 0) return -1; else return 0
P2: if (y > 0) return 1; else if (y <= 0) return -1;
```

By exploiting the renaming `x <-> y`, DSE could identify more paths to produce:

```
x renamed into y, P1: (x == 0), RETURN=0, P2: (y == 0), RETURN=-1
```

9 Future work

In this paper, we introduced the theoretical framework of differential operational semantics to give a formal meaning to (qualitative) differences between programs. A term of a difference language is a sound syntactic and declarative representation of a difference of behaviors between two programs if its interpretation by an oracle produces a relation between configurations that actually appear in the reduction chains of the two programs. Differences and difference languages can be composed which make it possible for complex semantic differences to be expressed in terms of more atomic semantic differences. Our first experiments in mechanizing this framework in the Coq proof assistant suggest that it is an appropriate foundational framework for certification.

The study of differential operational semantics is at an early stage: we now list several challenges that should in our opinion be tackled both on the theoretical and practical sides.

Difference languages over realistic programming languages Our experimentation on the Imp programming language was simple enough for us to focus on the design of the theoretical framework. As claimed earlier, we conjecture that our general definitions will be adapted to more realistic languages like functional and object-oriented languages as long as they are deterministic. Of course, new technical devices will be needed to design interesting difference languages and their definitions will probably be at a higher level of complexity than the ones for Imp. We are especially interested in designing difference languages for languages equipped with contract assertions because, as already noticed in existing work [23, 5, 27], formally capturing the semantic evolution of programming interfaces (API) is a key aspect to build useful tools for developpers. Another challenge is to extend our framework to deal with non determinism while preserving an interpretation of oracles that remains understandable to programmers.

Taxonomy of difference languages The difference languages on Imp presented here were meant to illustrate key aspects of the framework. One challenge is to invent more sophisticated difference languages and to have a way to classify them in a systematic way so that one can determine if a difference language is really new or if it is subsumed by an existing one.

Decision procedures and automatic analysis Our main practical motivation is to build (preferably certified) tools to give a qualitative account on the impact of changes on program implementations, interfaces and specifications. To achieve such goals, the challenge is to design difference inference algorithms over languages that actually correspond to the semantic patches programmers have in mind and that remains responsive in practice. Techniques and heuristics imported from differential static analysis will certainly be helpful.

Bibliography

[1] Abadi, M., Lamport, L., Lamport, L., Abadi, M.: The existence of refinement mappings. In: Proceedings of the 3rd Annual Symposium on Logic in Computer Science. pp. 165–175 (July 1988)

[2] Asperti, A.: The intensional content of rice's theorem. In: ACM SIGPLAN Notices. vol. 43, pp. 113–119. ACM (2008)

[3] Benton, N.: Simple relational correctness proofs for static analyses and program transformations (revised, long version) (2005)

[4] Bobot, F., Filliâtre, J.C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 platform 0.81 (Mar 2013), tutorial and Reference Manual

[5] Brunel, J., Doligez, D., Hansen, R.R., Lawall, J.L., Muller, G.: A foundation for flow-based program matching: using temporal logic and model checking. In: Acm Sigplan Notices. vol. 44-1, pp. 114–126. ACM (2009)

[6] Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K.: A theory of changes for higher-order languages: Incrementalizing Λ-calculi by static differentiation. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 145–155. PLDI '14, ACM, New York, NY, USA (2014)

[7] Cohen, J.: A Correct Refactoring Operation to Rename Global Variables in C Programs. Research report, LINA-University of Nantes (Dec 2015)

[8] Ferns, N., Panangaden, P., Precup, D.: Bisimulation metrics for continuous markov decision processes. SIAM Journal on Computing 40(6), 1662–1714 (2011)

[9] Girka, T., Mentré, D., Régis-Gianas, Y.: A mechanically checked generation of correlating programs directed by structured syntactic differences. In: International Symposium on Automated Technology for Verification and Analysis. pp. 64–79. Springer (2015)

[10] Gordon, A.D., Hankin, P.D., Lassen, S.B.: Compilation and equivalence of imperative objects. In: International Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 74–87. Springer (1997)

[11] Hawblitzel, C., Kawaguchi, M., Lahiri, S., Rebelo, H.: Mutual summaries: Unifying program comparison techniques. Tech. rep. (August 2011)

[12] Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing (preliminary report). In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 344–352. ACM (1989)

[13] Mardare, R., Priami, C., Quaglia, P., Vagin, O.: Model checking biological systems described using ambient calculus. In: International Conference on Computational Methods in Systems Biology. pp. 85–103. Springer (2004)

[14] Martin-Lef, P.: Intuitionistic type theory (1984)

[15] Milner, R.R.: A calculus of communicating systems. Lecture notes in computer science, Springer-Verlag, Berlin, New York (1980)

[16] Mimram, S., Di Giusto, C.: A categorical theory of patches. Electronic Notes in Theoretical Computer Science 298, 283–307 (2013)

[17] Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: ACM SIGPLAN Notices. vol. 49-10, pp. 811–828. ACM (2014)

[18] Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. pp. 226–237. ACM (2008)

[19] Pitts, A.M.: Operationally-based theories of program equivalence. Semantics and Logics of Computation 14 (1997)

[20] Plotkin, G.D.: A structural approach to operational semantics (1981)

[21] Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 669–685. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011)

[22] Shuvendu Lahiri, Kenneth McMillan, C.H.: Differential assertion checking. ACM (August 2013)

[23] Shuvendu Lahiri, Chris Hawblitzel, M.K.a.H.R.: Symdiff: A language-agnostic semantic diff tool for imperative programs. Springer (July 2012)

[24] Shuvendu Lahiri, Kapil Vaswani, T.H.: Differential static analysis: Opportunities, applications, and challenges. Association for Computing Machinery, Inc. (November 2010)

[25] Soares, G., Gheyi, R., Massoni, T.: Automated behavioral testing of refactoring engines. IEEE Transactions on Software Engineering 39(2), 147–162 (2013)

[26] Strichman, O., Godlin, B.: Regression verification-a practical way to verify programs. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 496–501. Springer (2005)

[27] Yi, J., Qi, D., Tan, S.H., Roychoudhury, A.: Expressing and checking intended changes via software change contracts. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 1–11. ACM (2013)

A Appendix

A.1 Step function for Imp

$$
\begin{aligned}
\mathsf{step}(M, \mathbf{skip}; \kappa) &= (M, \kappa) \\
\mathsf{step}(M, x = e; \kappa) &= (M[x := n], \kappa) \\
&\quad \text{where } M \vdash e \Downarrow n \\
\mathsf{step}(M, (C_1; C_2); \kappa) &= (M, C_1; (C_2; \kappa)) \\
\mathsf{step}(M, \mathbf{if}\ (b)\ C_1\ \mathbf{else}\ C_2; \kappa) &= (M, C_1; \kappa) \\
&\quad \text{where } M \vdash b \Downarrow \mathbf{true} \\
\mathsf{step}(M, \mathbf{if}\ (b)\ C_1\ \mathbf{else}\ C_2; \kappa) &= (M, C_2; \kappa) \\
&\quad \text{where } M \vdash b \Downarrow \mathbf{false} \\
\mathsf{step}(M, \mathbf{while}\ (b)\ C; \kappa) &= (M, C; \mathbf{while}\ (b)\ C; \kappa) \\
&\quad \text{where } M \vdash b \Downarrow \mathbf{true} \\
\mathsf{step}(M, \mathbf{while}\ (b)\ C; \kappa) &= (M, \kappa) \\
&\quad \text{where } M \vdash b \Downarrow \mathbf{false}
\end{aligned}
$$