Analysis of Software Patches Using Numerical Abstract Interpretation

David Delmas, Antoine Miné

Analysis of Software Patches Using Numerical Abstract Interpretation*

David Delmas[1,2] and Antoine Miné[2,3]

[1] Airbus Operations S.A.S., 316 route de Bayonne, 31060 Toulouse Cedex 9, France
[2] Sorbonne Université, CNRS, LIP6, 75005 Paris, France
[3] Institut universitaire de France, 1 rue Descartes, 75231 Paris Cedex 5, France
david.delmas@airbus.com, antoine.mine@lip6.fr

Abstract. We present a static analysis for software patches. Given two syntactically close versions of a program, our analysis can infer a semantic difference, and prove that both programs compute the same outputs when run on the same inputs. Our method is based on abstract interpretation, and parametric in the choice of an abstract domain. We focus on numeric properties only. Our method is able to deal with unbounded executions of infinite-state programs, reading from infinite input streams. Yet, it is limited to comparing terminating executions, ignoring non terminating ones.
We first present a novel concrete collecting semantics, expressing the behaviors of both programs at the same time. Then, we propose an abstraction of infinite input streams able to prove that programs that read from the same stream compute equal output values. We then show how to leverage classic numeric abstract domains, such as polyhedra or octagons, to build an effective static analysis. We also introduce a novel numeric domain to bound differences between the values of the variables in the two programs, which has linear cost, and the right amount of relationality to express useful properties of software patches.
We implemented a prototype and experimented on a few small examples from the literature. Our prototype operates on a toy language, and assumes a joint syntactic representation of two versions of a program given, which distinguishes between common and distinctive parts.

Introduction

The problem of proving the functional equivalence of programs, or program parts, is fundamental [7]. It aims at comparing the behaviors of two programs running in the same environment, *i.e.* their input-output relationships. In this paper, we describe a static analysis which aims at inferring that two syntactically close versions of a program compute equal outputs, when run on equal inputs.

The main application of this analysis is regression verification [8]: prove that a program change does not add any undesirable behavior. Take, for instance, the

* This work is performed as part of a collaborative partnership between Sorbonnne Université / CNRS (LIP6) and Airbus. This work is partially supported by the European Research Council under the Consolidator Grant Agreement 681393 – MOPSA.

```
172 172    /* Like fstatat, but cache the result.  If ST->st_size is -1, the
173 173       status has not been gotten yet.  If less than -1, fstatat failed
174     -     with errno == -1 - ST->st_size.  Otherwise, the status has already
    174 +     with errno == ST->st_ino.  Otherwise, the status has already
175 175       been gotten, so return 0.  */
176 176    static int
177 177    cache_fstatat (int fd, char const *file, struct stat *st, int flag)
178 178    {
179 179      if (st->st_size == -1 && fstatat (fd, file, st, flag) != 0)
180     -      st->st_size = -1 - errno;
    180 +      {
    181 +        st->st_size = -2;
    182 +        st->st_ino = errno;
    183 +      }
181 184      if (0 <= st->st_size)
182 185        return 0;
183     -    errno = -1 - st->st_size;
    186 +    errno = (int) st->st_ino;
184 187      return -1;
185 188    }
```

Fig. 1. Patch on remove.c of Coreutils (between v6.10 and v6.11)

commit shown on Fig. 1, extracted from a revision control repository of the GNU core utilities. It describes a change in a library implementing core functions for removing files and directories, and used by the POSIX rm command. The main function of this library uses the POSIX fstatat function to read information on the file to delete. As the same status information is needed in several contexts, the library implements a caching mechanism. At initialization, the main function calls a cache_stat_init function, which initializes the st_size field of the stat structure $*$st to -1. Then, it calls the cache_fstatat function shown on Fig. 1 repeatedly, whenever status information is needed. Indeed, cache_fstatat caches the results of the fstatat function. In revision v6.10 of Coreutils, this function used the st_size field of the stat structure $*$st to store information on the error value returned by fstatat upon the first call. It did it in a way that ensures that st_size <0 whenever errno>0, so as to use the sign of st_size upon subsequent calls, to distinguish between successful and erroneous executions. This scheme works for operating systems where errno is always set to positive values. However, some systems, such as BeOS [1] and Haiku [2], allow for negative errno values. The fix displayed on Fig. 1 aims at accommodating such systems. It consists in storing errno directly in the st_ino field of the stat structure.

On this example, non regression verification amounts to proving that the behavior of the main function of the library is unchanged on systems with only positive error values. This is, indeed, validated by our analysis. The analyzed source code includes a stub variable for errno, and stub code for the fstatat function. The stub for fstatat updates errno with a non-deterministic value, ranging

```
1   for (c=0; c<n; c++) cache_stat_init (&file[c].st);
2
3   while ((c=getchar()) >= 0 && c < n)
4     r = cache_fstatat (AT_FDCWD, file[c].name, &file[c].st,
          AT_SYMLINK_NOFOLLOW);
```

Fig. 2. Execution environments for cache_fstatat

over positive integers. Note that a separate analysis of the cache_fstatat function, as opposed to an analysis of the whole library, makes it necessary to model its possible execution environments with an unbounded loop, calling cache_fstatat an arbitrary number of times, with parameters taken from an arbitrary sequence of file names and stat structures. This unbounded sequence is modeled, in practice, using an unbounded number of reads from an input stream. Fig. 2 shows an example for n files, where n may be unbounded.

More generally, we are interested in analyzing patches of programs reading an unbounded number of input values, e.g. programs reading from file or I/O streams, and embedded reactive software with internal state, which no related work addresses. Or goal is to prove that the original and patched versions of such programs compute equal outputs, when run with the same sequence of inputs. We therefore model streams directly in the semantics on which our analysis is based (see section 1).

Running example In the following, we sketch our approach to the analysis of semantic differences between two syntactically similar programs P_1 and P_2. We are interested in proving that P_1 and P_2 compute equal outputs when run on equal inputs. P_1 and P_2 are represented together in the syntax of a so-called double program P. Simple programs P_1 and P_2 are referred to as the left and right projections of P. Fig. 3 shows the Unchloop example, taken from [24], and translated into our syntax of double programs. The $\|$ symbol is used to represent syntactic difference. It is available at expression, condition, and statement levels in our syntax for double programs. For instance at line 3, $c \leftarrow 1 \parallel 0$ means $c \leftarrow 1$ for P_1, and $c \leftarrow 0$ for P_2. In contrast, line 4 means $i \leftarrow 0$ for both P_1 and P_2.

Let us describe the example program. Both versions P_1 and P_2 read inputs in the range $[-1000, 1000]$ into a and b at lines 1 and 2. At line 3, the counter c is being initialised with value 1 for program P_1, and value 0 for program P_2. Then, both programs add a times the value of b to c in a loop. Finally, they both store the result into r at line 9: c for P_1, c+1 for P_2. The assertion at line 10 expresses the property we would like to check: if both programs reach it, then they should have computed equal values for r.

We assume here that both programs read the same input value in a, and the same input value in b. More generally, the semantics of P is parameterized by a (possibly infinite) sequence of input values, and we wish to prove that, given the same sequence of input values, P_1 and P_2 have the same result in r.

```
1 :   a ← input(−1000, 1000);
2 :   b ← input(−1000, 1000);
3 :   c ← 1 ‖ 0;
4 :   i ← 0;
5 :   while (i < a) {
6 :     c ← c + b;
7 :     i ← i + 1;
8 :   }
9 :   r ← c ‖ c + 1;
10 :  assert_sync(r);
```

Fig. 3. Unchloop example

```
1 : x ← input(−100, 100);
2 : if (x < 0) x ← −1;
3 : else {
4 :     if (x ≥ 2 ‖ x ≥ 4) {} // x > 4 in original paper
5 :     else {
6 :       while (i = 2) x ← 2;
7 :       x ← 3;
8 :     }
9 :   }
10 :  assert_sync(x); // x = 2 ignored
```

Fig. 4. Modified [24, Fig. 2] example

The assertion at line 10 of our example is thus valid. It is, indeed, validated by our analysis.

Limitations Our analysis is based on abstractions of a concrete collecting semantics which will be presented in section 1. This semantics relates pairs of terminating executions of projections of a double program. It is suitable to prove a number of properties, including that two terminating programs starting from equal initial states will produce equal outputs, a notion called partial equivalence in [8]. In contrast, an analysis based on this collecting semantics will fail to report differences between pairs of executions where at least one of the programs does not terminate. For instance, in the example on Fig. 4, our analysis does not report any difference between P_1 and P_2, although P_1 terminates on input $x = 2$, and P_2 does not.

As opposed to [21,22], which develop algorithms to automate the construction of a correlating program $P_1 \bowtie P_2$, on which to run the static analysis, we assume for now the joint representation of P_1 and P_2 given, as part of a double program in our toy language.

Related work [11] pioneered the field of semantic differencing between two versions of a procedure by comparing dependencies between input and output variables. Symbolic execution methods [23,24,19] have proposed analysis techniques

for programs with small state space and bounded loops, which may support modular regression verification. On the contrary, we can handle programs with unbounded loops and an infinite number of execution paths, like the example of Figs. 1 and 2. Some approaches [16] combine symbolic execution and program analysis techniques to improve the coverage of patches with tests suites, but such testing coverage criteria bring no formal guarantee of correctness, unlike our method.

RVT [8] and SymDiff [14,15] combine two versions of the same program, with equality constraints on their inputs, and compile equivalence properties into verification conditions to be checked by SMT solvers. On the contrary, we rely on abstract domains to infer equivalence properties.

The DIZY [21,22] tool leverages numerical abstract interpretation to establish equivalence under abstraction. In particular, the authors give a semi-formal description of an operational concrete trace semantics. This semantics is not defined by induction on the syntax, and does not support streams. Our main contribution, with respect to this work, is a novel, fully formalized, denotational concrete collecting semantics by induction on the syntax, which can deal with programs reading from infinite input streams, and a novel numeric domain to bound differences between the values of the variables in the two programs. Another difference is that [21,22] rely on program transformations to build a correlating program, which they analyze according to simple program semantics, while our semantics is defined for double programs directly.

The Fluctuat [17,9] static analyser compares the real and floating-point semantics of numeric programs to bound errors in floating-point computations. The authors use the zonotope abstract domain to bound the difference between real and floating-point values. Like in our concrete semantics, they also address unstable test analysis [10].

Contributions The main contributions of this work are:

- We present a novel concrete collecting semantics, expressing the behaviors of two versions of a program at the same time. This semantics deals with programs reading from unbounded input streams.
- We propose an abstraction of infinite input streams able to prove that programs that read from the same stream compute equal output values.
- We introduce a novel numeric domain to bound differences between the values of the variables in the two programs, which has linear cost, and the right amount of relationality to express useful properties of software patches.
- We implemented a prototype static analyzer which exhibits significant speedups with respect to previous works.

We build on previous work [6]. The main contributions of the current paper, with respect to this work, is a formal treatment of infinite input streams, in the concrete and abstract semantics.

The paper is organised as follows. Section 1 formalizes the concrete collecting semantics, and illustrates it on the example from Fig. 3. Section 2 describes

$$
\begin{array}{lll}
\mathit{stat} ::= & V \leftarrow \mathit{expr} & V \in \mathcal{V} \\
| & V \leftarrow \textbf{input}(a, b) & a, b \in \mathbb{R} \\
| & \textbf{if } \mathit{cond} \textbf{ then } \mathit{stat} \textbf{ else } \mathit{stat} & \\
| & \textbf{while } \mathit{cond} \textbf{ do } \mathit{stat} & \\
| & \mathit{stat}; \mathit{stat} & \\
| & \textbf{skip} &
\end{array}
$$

(a) Simple statements

$$
\begin{array}{lll}
\mathit{expr} ::= & V & V \in \mathcal{V} \\
| & c & c \in \mathbb{R} \\
| & -\mathit{expr} & \\
| & \mathit{expr} \diamond \mathit{expr} & \diamond \in \{+, -, \times, /\} \\
| & \textbf{rand}(a, b) & a, b \in \mathbb{R} \\
\mathit{cond} ::= & \mathit{expr} \bowtie \mathit{expr} & \bowtie \in \{\leq, \geq, =, \neq, <, >\} \\
| & \neg \mathit{cond} & \\
| & \mathit{cond} \diamond \mathit{cond} & \diamond \in \{\wedge, \vee\}
\end{array}
$$

(b) Simple expressions and conditions

$$
\begin{array}{lll}
\mathit{dstat} ::= & \mathit{stat} & \\
| & \mathit{stat} \parallel \mathit{stat} & \\
| & V \leftarrow \mathit{dexpr} & V \in \mathcal{V} \\
| & \textbf{assert\_sync}(V) & \\
| & \mathit{dstat}; \mathit{dstat} & \\
| & \textbf{if } \mathit{dcond} \textbf{ then } \mathit{dstat} \textbf{ else } \mathit{dstat} & \\
| & \textbf{while } \mathit{dcond} \textbf{ do } \mathit{dstat} &
\end{array}
$$

(c) Double statements

$$
\begin{array}{lll}
\mathit{dexpr} ::= & \mathit{expr} & \\
| & \mathit{expr} \parallel \mathit{expr} & \\
\mathit{dcond} ::= & \mathit{cond} & \\
| & \mathit{cond} \parallel \mathit{cond} &
\end{array}
$$

(d) Double expressions and conditions

Fig. 5. Syntax of simple and double programs

the abstract semantics, discusses the choice of numeric abstract domains, and introduces a novel numeric domain. Section 3 presents experimental results with a prototype implementation. Section 4 concludes.

1 Syntax and concrete semantics

Following the standard approach to abstract interpretation [4], we developed a concrete collecting semantics for a toy While language for double programs. The $\parallel$ operator may occur anywhere in the parse tree, to denote syntactic differences between the left and right projections of a double program. However, $\parallel$ operators cannot be nested: a double program only describes a pair of programs.

Given double program P with variables in $\mathcal{V}$, consider its left (resp. right) projection $P_1 = \pi_1(P)$ (resp. $P_2 = \pi_2(P)$), where π_1 (resp. π_2) is a projection operator defined by induction on the syntax, keeping only the left (resp. right) side of $\parallel$ symbols. For instance, $\pi_1(c \leftarrow 1 \parallel 0) = c \leftarrow 1$, and $\pi_2(c \leftarrow 1 \parallel 0) = c \leftarrow 0$, while $\pi_1(i \leftarrow 0) = i \leftarrow 0 = \pi_2(i \leftarrow 0)$.

1.1 Simple programs

P_1 and P_2 are simple programs, with concrete memory states in $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{R}$. Let $k \in \{1; 2\}$. The syntax of simple program P_k is standard. Statements *stat* are presented in Fig. 5(a). They are built on top of numeric expressions *expr* and Boolean conditions *cond*, defined in Fig. 5(b). To define the semantics of simple program P_k, we leverage standard, relational, input-output semantics, defined by induction on the syntax, in denotational style. Given $\mathbb{E}[\![e]\!] \in \mathcal{E} \to \mathcal{P}(\mathbb{R})$ for non-deterministic expression $e \in \mathit{expr}$, and $\mathbb{C}[\![c]\!] \in \mathcal{E} \to \mathcal{P}(\{\text{true}, \text{false}\})$ for condition $c \in \mathit{cond}$, we let $\mathbb{S}[\![s]\!]$ describe the relation between input and output states of statement $s \in \mathit{stat}$. Because of the **input** command, which

reads some input stream, $\mathbb{S}[\![s]\!]$ is parameterised by a sequence of values, and program states record the current index in this sequence. Note that this sequence has to be infinite: indeed, due to non-determinism, the concrete semantics maps every input stream to a (possibly infinite) set of executions, which can execute an unbounded number of input statements. Therefore $\mathbb{S}[\![s]\!] \in \mathbb{R}^\omega \to \mathcal{P}(\mathcal{E}' \times \mathcal{E}')$, where $\mathcal{E}' \triangleq \mathcal{E} \times \mathbb{N}$, and:

$$\mathbb{S}[\![V \leftarrow \textbf{input}(a,b)]\!]\sigma \triangleq \{\, ((\rho,n),(\rho[V \mapsto \sigma_n], n+1)) \mid (\rho,n) \in \mathcal{E}' \wedge a \leq \sigma_n \leq b \,\}$$

Note that we model one input stream only, but the generalization to several input streams is obvious. We do not display the semantics for other commands, as the semantics for assignments and tests are standard for memory environments, and leave indexes unchanged. For instance, $\mathbb{S}[\![V \leftarrow e]\!]\sigma \triangleq \{\, ((\rho,n),(\rho[V \mapsto v], n)) \mid (\rho,n) \in \mathcal{E}' \wedge v \in \mathbb{E}[\![e]\!]\rho \,\}$.

1.2 Double programs

We then lift the semantics $\mathbb{S}$ to double programs. As P_1 and P_2 have concrete states in $\mathcal{E}'$, P has concrete states in $\mathcal{D}' \triangleq \mathcal{E}' \times \mathcal{E}'$. The syntax of double statements *dstat* is shown in Fig. 5(c). They are built on top of double expressions *dexpr* and double conditions *dcond*, defined in Fig. 5(d). The semantics of a double statement $s \in$ *dstat*, denoted $\mathbb{D}[\![s]\!] \in \mathbb{R}^\omega \to \mathcal{P}(\mathcal{D}' \times \mathcal{D}')$, describes the relation between input and output states of s, which are pairs of states of simple programs, for a given shared sequence of input values. The definition for $\mathbb{D}[\![s]\!]$ is shown on Fig. 6, in relational style. It is defined by induction on the syntax, so as to allow for modular, joint analyses of double programs that maintain input-output relations on the variables. Note that $\mathbb{D}$ is parametric in $\mathbb{S}$.

The semantics for the empty program is the diagonal, identity relation $\Delta_{\mathcal{D}'}$. The semantics $\mathbb{D}[\![s_1 \parallel s_2]\!]$ for the composition of two syntactically different statements reverts to the pairing of the simple program semantics of individual simple statements s_1 and s_2. Note that $\mathbb{D}[\![s_1 \parallel s_2]\!]\sigma = \mathbb{D}[\![s_1 \parallel \textbf{skip}]\!]\sigma \fatsemi \mathbb{D}[\![\textbf{skip} \parallel s_2]\!]\sigma$ for any $\sigma \in \mathbb{R}^\omega$, where we use the symbol $\fatsemi$ to denote the left composition of relations: $R_1 \fatsemi R_2 \triangleq \{\, (x,z) \mid \exists y : (x,y) \in R_1 \wedge (y,z) \in R_2 \,\}$. The semantics for assignments of double expressions $\mathbb{D}[\![V \leftarrow e_1 \parallel e_2]\!]$ (different expressions to the same variable) is defined using this construct. The interest of double expressions in the syntax is to allow for simple symbolic simplifications in later abstraction steps, when computing differences between expressions assigned to a variable. The semantics of **assert_sync**(V) statements asserts that the left and right projections of a double program agree on the value of variable V. The semantics for the sequential composition of statements boils down to the composition of the semantics of individual statements. The semantics for selection statements relies on the filter $\mathbb{F}[\![c_1 \parallel c_2]\!]$ to distinguish between cases where both projections agree on the value of the controlling expression, and cases where they do not (*a.k.a.* unstable tests). There are two stable and two unstable test cases, according to the evaluations of the two conditions. The semantics for stable test cases is standard. The semantics for unstable test cases is defined by composing the left

$\mathbb{D}[\![\, dstat \,]\!] \in \mathbb{R}^\omega \to \mathcal{P}(\mathcal{D}' \times \mathcal{D}')$

$$
\begin{array}{ll}
\mathbb{D}[\![\, \textbf{skip} \,]\!]\sigma & \triangleq \Delta_{\mathcal{D}'} \\
\mathbb{D}[\![\, s_1 \parallel s_2 \,]\!]\sigma & \triangleq \{\, ((i_1,i_2),(o_1,o_2)) \mid (i_1,o_1) \in \mathbb{S}[\![\, s_1 \,]\!]\sigma \wedge (i_2,o_2) \in \mathbb{S}[\![\, s_2 \,]\!]\sigma \,\} \\
\mathbb{D}[\![\, V \leftarrow e_1 \parallel e_2 \,]\!]\sigma & \triangleq \mathbb{D}[\![\, V \leftarrow e_1 \parallel V \leftarrow e_2 \,]\!]\sigma \\
\mathbb{D}[\![\, V \leftarrow e \,]\!]\sigma & \triangleq \mathbb{D}[\![\, V \leftarrow e \parallel V \leftarrow e \,]\!]\sigma \\
\mathbb{D}[\![\, V \leftarrow \textbf{input}(a,b) \,]\!]\sigma & \triangleq \mathbb{D}[\![\, V \leftarrow \textbf{input}(a,b) \parallel V \leftarrow \textbf{input}(a,b) \,]\!]\sigma \\
\mathbb{D}[\![\, \textbf{assert\_sync}(V) \,]\!]\sigma & \triangleq \{\, (((\rho_1,n_1),(\rho_2,n_2)),((\rho_1,n_1),(\rho_2,n_2))) \mid \rho_1(V) = \rho_2(V) \,\} \\
\mathbb{D}[\![\, s\,;\,t \,]\!]\sigma & \triangleq \mathbb{D}[\![\, s \,]\!]\sigma \fatsemi \mathbb{D}[\![\, t \,]\!]\sigma \\
\mathbb{D}[\![\, \textbf{if } c_1 \parallel c_2 \textbf{ then } s \textbf{ else } t \,]\!]\sigma & \triangleq \mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \fatsemi \mathbb{D}[\![\, s \,]\!]\sigma \\
& \quad \cup \mathbb{F}[\![\, c_1 \parallel \neg c_2 \,]\!] \fatsemi \mathbb{D}[\![\, \pi_1(s) \parallel \textbf{skip} \,]\!]\sigma \fatsemi \mathbb{D}[\![\, \textbf{skip} \parallel \pi_2(t) \,]\!]\sigma \\
& \quad \cup \mathbb{F}[\![\, \neg c_1 \parallel c_2 \,]\!] \fatsemi \mathbb{D}[\![\, \pi_1(t) \parallel \textbf{skip} \,]\!]\sigma \fatsemi \mathbb{D}[\![\, \textbf{skip} \parallel \pi_2(s) \,]\!]\sigma \\
& \quad \cup \mathbb{F}[\![\, \neg c_1 \parallel \neg c_2 \,]\!] \fatsemi \mathbb{D}[\![\, t \,]\!]\sigma \\
\mathbb{D}[\![\, \textbf{if } c \textbf{ then } s \textbf{ else } t \,]\!]\sigma & \triangleq \mathbb{D}[\![\, \textbf{if } c \parallel c \textbf{ then } s \textbf{ else } t \,]\!]\sigma \\
\mathbb{D}[\![\, \textbf{while } c_1 \parallel c_2 \textbf{ do } s \,]\!]\sigma & \triangleq (\mathrm{lfp}\ H) \fatsemi \mathbb{F}[\![\, \neg c_1 \parallel \neg c_2 \,]\!] \\
\mathbb{D}[\![\, \textbf{while } c \textbf{ do } s \,]\!]\sigma & \triangleq \mathbb{D}[\![\, \textbf{while } c \parallel c \textbf{ do } s \,]\!]\sigma
\end{array}
$$

where $\mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \triangleq \{\, (((\rho_1,n_1),(\rho_2,n_2)),((\rho_1,n_1),(\rho_2,n_2))) \mid \mathrm{true} \in \mathbb{C}[\![\, c_1 \,]\!]\rho_1 \cap \mathbb{C}[\![\, c_2 \,]\!]\rho_2 \,\}$

and $H(R) \triangleq \Delta_{\mathcal{D}'} \cup R \fatsemi \begin{pmatrix} \mathbb{F}[\![\, c_1 \parallel c_2 \,]\!] \fatsemi \mathbb{D}[\![\, s \,]\!]\sigma & \cup \\ \mathbb{F}[\![\, c_1 \parallel \neg c_2 \,]\!] \fatsemi \mathbb{D}[\![\, \pi_1(s) \parallel \textbf{skip} \,]\!]\sigma & \cup \\ \mathbb{F}[\![\, \neg c_1 \parallel c_2 \,]\!] \fatsemi \mathbb{D}[\![\, \textbf{skip} \parallel \pi_2(s) \,]\!]\sigma & \end{pmatrix}$

Fig. 6. Denotational concrete semantics of double programs

restriction of the left projection $\pi_1(s) \parallel \textbf{skip}$ and the right restriction of the right projection $\textbf{skip} \parallel \pi_2(t)$ of the **then** s and **else** t branches. Intuitively, $\pi_1(s) \parallel \textbf{skip}$ means that the left projection of the double program executes s, while the right projection of the double program does nothing. The semantics for (possibly unbounded) iteration statements is defined using the least fixpoint of a function defined similarly.

Note that the semantics $\mathbb{D}[\![\, V \leftarrow \textbf{input}(a,b) \,]\!]$ of input statements is different from the semantics $\mathbb{D}[\![\, V \leftarrow \textbf{rand}(a,b) \,]\!]$ of non-deterministic assignments. The latter entails no relationship between the values read by the two projections of a double program, besides the fact that they range in the same interval. On the contrary, the former reads from a shared input stream σ, hence the left and right projections P_1 and P_2 read equal values if their input indexes n_1 and n_2 are equal. This is the case when P_1 and P_2 have called **input** equal numbers of times. On the contrary, if one projection, say P_1, has called **input** more often than the other, then P_1 is ahead of P_2 in the stream, and the two projections are desynchronized. Nonetheless, they may resynchronize later if P_2 catches up with P_1, hence read equal values again. Also, owing to the semantics $\mathbb{S}[\![\, V \leftarrow \textbf{input}(a,b) \,]\!]$ of simple input statements, $\textbf{input}(a,b)$ returns only if the input value at the current index is in the range $[a,b]$. Therefore, it should be considered a semantic error if P_1 and P_2 use different ranges $[a_1,b_1] \neq [a_2,b_2]$

to read the input at the same index. For the sake of simplicity, we do not check this in our semantics (altough our implementation performs this check).

The presence of both **input** and **rand** primitives makes the semantics very expressive, and useful for modeling many practical problems. Non-determinism allows to abstract unknown parts of a program: for instance, **rand**$(0, 10)$ is a sound stub for $f()$, when function f is only known to return values between 0 and 10. Also, combining **input** and **rand** allows to model information flow problems. For instance, the $\mathbb{D}$ semantics distinguishes the two programs 21(a) and 21(b) shown on Fig 21, which will be presented in the conclusion.

1.3 Properties of interest

We wish to prove the functional equivalence of the left and right projections of a given double program $P \in \mathit{dstat}$, restricted to a set of distinguished variables $\mathcal{V}_0 \in \mathcal{P}(\mathcal{V})$, specified with the **assert_sync** primitive. Let $I_0 \triangleq \{((\lambda V.\, 0, 0), (\lambda V.\, 0, 0))\}$ be the singleton state with all variables and input indexes initialized to zero. The set of states reachable by P from I_0 with input stream σ is $(\mathbb{D}[\![\, P \,]\!]\sigma)I_0$. Therefore the property of interest may be formalized as:

$$\forall \sigma \in \mathbb{R}^\omega : \forall V \in \mathcal{V}_0 : \forall((\rho_1, n_1), (\rho_2, n_2)) \in (\mathbb{D}[\![\, P \,]\!]\sigma)I_0 : \rho_1(V) = \rho_2(V)$$

Coming back to our running example Unchloop on Fig. 3, the concrete semantics of the program from line 3 to 9 is displayed on Fig. 7, for any sequence of inputs $\sigma \in \mathbb{R}^\omega$. With the additional assumption that both program projections compute with equal inputs ($a_1 = a_2 = \sigma_0 \wedge b_1 = b_2 = \sigma_1$), ensured by the semantics of line 1 and 2, and the initial environment I_0, the two projections can be proved to compute equal values for r.

$$
\begin{array}{l}
\mathbb{D}[\![\, \mathit{Unchloop}_{3..9} \,]\!]\sigma = \\
\quad \{\, s_0, ((a_1, b_1, 1, 0, 1, n_1), (a_2, b_2, 0, 0, 1, n_2)) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid a_1 \leq 0 \wedge a_2 \leq 0 \wedge H_0 \,\} \\
\cup \{\, s_0, ((a_1, b_1, 1 + a_1 \times b_1, a_1, 1 + a_1 \times b_1, n_1), (a_2, b_2, 0, 0, 1, n_2)) \qquad\qquad\qquad\qquad \mid a_1 > 0 \wedge a_2 \leq 0 \wedge H_0 \,\} \\
\cup \{\, s_0, ((a_1, b_1, 1, 0, 1, n_1), (a_2, b_2, a_2 \times b_2, a_2, 1 + a_2 \times b_2, n_2)) \qquad\qquad\qquad\qquad \mid a_1 \leq 0 \wedge a_2 > 0 \wedge H_0 \,\} \\
\cup \{\, s_0, ((a_1, b_1, 1 + a_1 \times b_1, a_1, 1 + a_1 \times b_1, n_1), (a_2, b_2, a_2 \times b_2, a_2, 1 + a_2 \times b_2, n_2)) \mid a_1 > 0 \wedge a_2 > 0 \wedge H_0 \,\}
\end{array}
$$

where $s_0 \triangleq ((a_k, b_k, c_k, i_k, r_k), n_k)_{k \in \{1,2\}}$
and $H_0 \triangleq s_0 \in \mathbb{R}^4 \times \mathbb{N}$

Fig. 7. Concrete semantics of the Unchloop example from Fig. 3

Unfortunately, our concrete collecting semantics $\mathbb{D}$ is not computable in general. A particular difficulty of the Unchloop example is that the input-ouptut relation is non linear: $(a \leq 0 \Rightarrow r = 1) \wedge (a \geq 0 \Rightarrow r = 1 + a \times b)$. Hence, inferring such information is beyond classic numeric domains, such as polyhedra. We will provide a new analysis method which avoids resorting to more complex, non-linear numeric domains. An additional difficulty, not shown in the Unchloop example, is that the programs can read an unbounded number of values from their input stream.

$$(\mathbb{R}^\omega \to \mathcal{P}(\mathcal{D}' \times \mathcal{D}'), \dot{\subseteq}) \underset{\alpha_{\mathfrak{F}}}{\overset{\gamma_{\mathfrak{F}}}{\leftrightarrows}} (\mathcal{P}(\hat{\mathcal{D}} \times \hat{\mathcal{D}}), \subseteq)$$

$$\begin{aligned} \alpha_{\mathfrak{F}}(f) &\triangleq \{\, (\beta_\sigma(s), \beta_\sigma(s')) \,|\, (s, s') \in f(\sigma) \wedge \sigma \in \mathbb{R}^\omega \,\} \\ (\gamma_{\mathfrak{F}}(R))(\sigma) &\triangleq \{\, (s, s') \,|\, (\beta_\sigma(s), \beta_\sigma(s')) \in R \,\} \end{aligned}$$

where $\forall\, \sigma \in \mathbb{R}^\omega : \beta_\sigma \in \mathcal{D}' \to \hat{\mathcal{D}}$

$$\begin{aligned} \beta_\sigma(((\rho_1, n_1), (\rho_2, n_2))) &\triangleq (\rho_1, \rho_2, \delta, q) \\ \text{with} \quad \delta = n_2 - n_1 \wedge |q| = |\delta| \wedge \forall\, 0 \leq n < |\delta| &: q_n = \sigma_{\max\{n_1, n_2\} - n - 1} \end{aligned}$$

Fig. 8. Abstraction of shared input sequences with unbounded FIFO queues

2 Abstract semantics

We therefore tailor an abstract semantics suitable for the analysis of program differences.

2.1 Wrapping up infinite input sequences

A first observation is that we do not need to recall the whole input sequence $\sigma \in \mathbb{R}^\omega$ shared by the left and right projections P_1 and P_2 of a double program P. Indeed, we only aim at inferring equalities between the input values read by P_1 and P_2. We therefore only need to record, at any point in the analysis, the input subsequence that has been read by one program, but not the other one yet. This ensures that, when a program that has read less values than the other catches up with it, it reads the same values. Values read by both programs can be discarded, and values not read by any program do not need to be known in advance, as they can be chosen non-deterministically. This subsequence of input values read by one program only forms an (unbounded) FIFO queue, as inputs are read in order. We therefore abstract the input sequence σ, and indexes n_1 and n_2 of P_1 and P_2 in this sequence, defined in $\mathcal{D}'$, as the difference $\delta \triangleq n_2 - n_1$, and a FIFO queue of length $|\delta|$ in $\hat{\mathcal{D}} \triangleq \mathcal{E} \times \mathcal{E} \times \mathbb{Z} \times \mathbb{R}^\star$. This abstraction does not lose information. A formalization of this abstraction is shown on Fig. 8. Note that we use the symbol $\dot{\subseteq}$ to denote the pointwise lifting of $\subseteq$: $f \dot{\subseteq} f' \equiv \forall \sigma \in \mathbb{R}^\omega : f(\sigma) \subseteq f'(\sigma)$.

Proposition 1. *The pair $(\alpha_{\mathfrak{F}}, \gamma_{\mathfrak{F}})$ defined in Fig. 8 is a Galois isomorphism.*

Note that this abstraction includes some redundancy: indeed, it would be enough to record only the sign of δ, instead of its value, as its absolute value is given by the length of the queue. However, keeping the value simplifies subsequent abstraction steps.

Simple programs Starting from the concrete semantics $\mathbb{D}$, let us now formalize the semantics resulting from this first abstraction step. To start with, we first define the simple program semantics. The behaviors of the left and right projections P_1 and P_2 of a double program P depend on which is ahead in the input sequence, and which is behind. P_1 is ahead if $\delta < 0$, and P_2 is

$\hat{\mathbb{S}}_k[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{E}} \times \hat{\mathcal{E}}) \quad ; \quad k \in \{1, 2\}$

$$\hat{\mathbb{S}}_1[\![\, V \leftarrow \mathbf{input}(a, b) \,]\!] \triangleq \\ \{\, ((\rho, \delta, q), (\rho[V \mapsto \nu], \delta - 1, \nu \cdot q)) \mid (\rho, \delta, q) \in \hat{\mathcal{E}} \wedge \delta \leq 0 \wedge a \leq \nu \leq b \,\} \\ \cup \{\, ((\rho, \delta, q \cdot v), (\rho[V \mapsto v], \delta - 1, q)) \mid (\rho, \delta, q) \in \hat{\mathcal{E}} \wedge \delta > 0 \wedge a \leq v \leq b \,\}$$

$$\hat{\mathbb{S}}_2[\![\, V \leftarrow \mathbf{input}(a, b) \,]\!] \triangleq \\ \{\, ((\rho, \delta, q), (\rho[V \mapsto \nu], \delta + 1, \nu \cdot q)) \mid (\rho, \delta, q) \in \hat{\mathcal{E}} \wedge \delta \geq 0 \wedge a \leq \nu \leq b \,\} \\ \cup \{\, ((\rho, \delta, q \cdot v), (\rho[V \mapsto v], \delta + 1, q)) \mid (\rho, \delta, q) \in \hat{\mathcal{E}} \wedge \delta < 0 \wedge a \leq v \leq b \,\}$$

Fig. 9. Abstract semantics of simple programs P_1 and P_2 with unbounded queues

ahead if $\delta > 0$. Therefore, we need to particularize the simple program semantics $\hat{\mathbb{S}}_k[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{E}} \times \hat{\mathcal{E}})$, where $\hat{\mathcal{E}} \triangleq \mathcal{E} \times \mathbb{Z} \times \mathbb{R}^\star$, and $k \in \{1, 2\}$. Fig. 9 shows the semantics for $\hat{\mathbb{S}}_k[\![\, V \leftarrow \mathbf{input}(a, b) \,]\!]$. Note that we write $q \cdot q'$ to denote concatenation of queues q and q'. Intuitively, this semantics distinguishes between two cases:

1. If program P_k is ahead of the other program in the input sequence, or at the same point, then a new successful input read operation produces a fresh input value, and adds it at the head of the queue.
2. If program P_k is behind the other program in the input sequence, then a new successful input read operation retrieves the value at the tail of the queue.

In both cases, an input read operation is only successful if the value read matches the bounds specified for the **input** statement. We do not display the semantics for other commands, as the semantics for assignments and tests are standard for memory environments, and leave input index differences and queues unchanged. For instance, $\hat{\mathbb{S}}_k[\![\, V \leftarrow e \,]\!] \triangleq \{\, ((\rho, \delta, q), (\rho[V \mapsto v], \delta, q)) \mid (\rho, \delta, q) \in \hat{\mathcal{E}} \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$.

Double programs We then lift the semantics $\hat{\mathbb{S}}_1[\![\, s \,]\!]$ and $\hat{\mathbb{S}}_2[\![\, s \,]\!]$ to double programs. The definition of $\hat{\mathbb{D}}[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{D}} \times \hat{\mathcal{D}})$ is very similar to that of $\mathbb{D}[\![\, s \,]\!]$. It can be obtained by removing σ parameters from Fig. 6, except for the composition of syntactically different statements $\hat{\mathbb{D}}[\![\, s_1 \parallel s_2 \,]\!]$ and conditions $\hat{\mathbb{F}}[\![\, c_1 \parallel c_2 \,]\!]$ We thus only show the definitions of these relations on Fig. 10. Following the particularization of simple statement semantics, the semantics for double statements and conditions compose the semantics of their left and right projections $\hat{\mathbb{D}}_k[\![\, s_k \,]\!]$ and $\hat{\mathbb{F}}_k[\![\, c_k \,]\!]$, where $\hat{\mathbb{D}}_k$ and $\hat{\mathbb{F}}_k$ operate on simple statements and conditions only. Note that the order of the composition is arbitrary, and not significant, as $\hat{\mathbb{D}}_1[\![\, s \,]\!] \fatsemi \hat{\mathbb{D}}_2[\![\, t \,]\!] = \hat{\mathbb{D}}_2[\![\, t \,]\!] \fatsemi \hat{\mathbb{D}}_1[\![\, s \,]\!]$, and likewise for $\hat{\mathbb{F}}_1[\![\, c \,]\!]$ and $\hat{\mathbb{F}}_2[\![\, d \,]\!]$. Finally, we formalize the relation between the abstract semantics $\hat{\mathbb{D}}$ and the concrete collecting semantics $\mathbb{D}$.

Proposition 2. *$\hat{\mathbb{D}}$ is a sound and complete abstraction of $\mathbb{D}$: $\hat{\mathbb{D}} = \alpha_{\mathfrak{F}}(\mathbb{D})$.*

$\hat{\mathbb{D}}[\![s]\!] \in \mathcal{P}(\hat{\mathcal{D}} \times \hat{\mathcal{D}})$

$$\begin{aligned}
\hat{\mathbb{D}}[\![s_1 \parallel s_2]\!] &\triangleq \hat{\mathbb{D}}_1[\![s_1]\!] \fatsemi \hat{\mathbb{D}}_2[\![s_2]\!] \\
\hat{\mathbb{D}}_1[\![s]\!] &\triangleq \{\, ((\rho_1, \rho_2, \delta, q), (\rho_1', \rho_2, \delta', q')) \mid ((\rho_1, \delta, q), (\rho_1', \delta', q')) \in \hat{\mathbb{S}}_1[\![s]\!] \wedge \rho_2 \in \mathcal{E} \,\} \\
\hat{\mathbb{D}}_2[\![s]\!] &\triangleq \{\, ((\rho_1, \rho_2, \delta, q), (\rho_1, \rho_2', \delta', q')) \mid ((\rho_2, \delta, q), (\rho_2', \delta', q')) \in \hat{\mathbb{S}}_2[\![s]\!] \wedge \rho_1 \in \mathcal{E} \,\} \\
\\
\hat{\mathbb{F}}[\![c_1 \parallel c_2]\!] &\triangleq \hat{\mathbb{F}}_1[\![c_1]\!] \fatsemi \hat{\mathbb{F}}_2[\![c_2]\!] \\
\hat{\mathbb{F}}_k[\![c]\!] &\triangleq \{\, ((\rho_1, \rho_2, \delta, q), (\rho_1, \rho_2, \delta, q)) \mid (\rho_1, \rho_2, \delta, q) \in \hat{\mathcal{D}} \wedge \mathrm{true} \in \mathbb{C}[\![c]\!]\rho_k \,\} \;;\; k \in \{1; 2\}
\end{aligned}$$

Fig. 10. Abstract semantics of double programs with unbounded queues

$$(\mathcal{P}(\hat{\mathcal{D}} \times \hat{\mathcal{D}}), \subseteq) \underset{\alpha_p}{\overset{\gamma_p}{\leftrightarrows}} (\mathcal{P}(\hat{\mathcal{D}}_p \times \hat{\mathcal{D}}_p), \subseteq)$$

$$\begin{aligned}
\alpha_p(R) &\triangleq \{\, (\beta_p(s), \beta_p(s')) \mid (s, s') \in R \,\} \\
\gamma_p(R) &\triangleq \{\, (s, s') \mid (\beta_p(s), \beta_p(s')) \in R \,\}
\end{aligned}$$

where $\beta_p \in \hat{\mathcal{D}} \to \hat{\mathcal{D}}_p$

$$\beta_p((\rho_1, \rho_2, \delta, q)) \triangleq (\rho_1, \rho_2, \delta, \tilde{q}) \;\text{ with }\; \tilde{q}_n = \begin{cases} q_n & \textbf{if } 0 \leq n < |\delta| \\ 0 & \textbf{if } |\delta| \leq n < p \end{cases}$$

Fig. 11. Abstraction of FIFO queues to fixed length $p \geq 1$

2.2 Bounding input queues

The abstract semantics $\hat{\mathbb{D}}$ features unbounded queues. We aim at abstracting the concrete collecting semantics $\mathbb{D}$ in numeric domains, so we need to deal with a bounded number of variables. As it is also simpler to deal with a fixed number of variables, we parameterize our abstract semantics with some predetermined integer $p \geq 1$, used to define the lengths of abstract FIFO queues in domain $\hat{\mathcal{D}}_p \triangleq \mathcal{E} \times \mathcal{E} \times \mathbb{Z} \times \mathbb{R}^p$. Queues from $\hat{\mathcal{D}}$ are truncated whenever $|\delta| > p$, and padded with zeros whenever $|\delta| < p$. A formalization of this abstraction is shown on Fig. 11.

Proposition 3. *For all $p \geq 1$, the pair (α_p, γ_p) defined in Fig. 11 is a Galois embedding.*

Let $p \geq 1$. Starting from semantics $\hat{\mathbb{D}}$, we now give a formal definition for the abstract double program semantics $\hat{\mathbb{D}}^p$ resulting from this second abstraction step.

Simple programs To this aim, we first define the semantics $\hat{\mathbb{S}}^p_k[\![s]\!] \in \mathcal{P}(\hat{\mathcal{E}}_p \times \hat{\mathcal{E}}_p)$ of simple programs, where $\hat{\mathcal{E}}_p \triangleq \mathcal{E} \times \mathbb{Z} \times \mathbb{R}^p$, and $k \in \{1, 2\}$. Fig. 12 shows the semantics of $\hat{\mathbb{S}}^p_1[\![V \leftarrow \textbf{input}(a, b)]\!]$. *Mutatis mutandis*, the case of $\hat{\mathbb{S}}^p_2$ is similar. Intuitively, this semantics distinguishes between three cases:

1. If program P_k is ahead of the other program in the input sequence, or at the same point, then a new successful input read operation produces a fresh

$\hat{\mathbb{S}}^p_1[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{E}}_p \times \hat{\mathcal{E}}_p)$

$$\begin{aligned}
\hat{\mathbb{S}}^p_1[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] \triangleq\ & \\
& \left\{\, ((\rho_1,\delta,q\cdot v),(\rho_1[V_1 \mapsto \nu],\delta-1,\nu\cdot q)) \;\middle|\; (\rho_1,\delta,q)\in\mathcal{E}_{p-1} \wedge \delta \le 0 \wedge a \le \nu \le b \wedge v \in \mathbb{R} \,\right\} \\
\cup & \left\{\, ((\rho_1,\delta,q\cdot v\cdot r),(\rho_1[V_1 \mapsto v],\delta-1,q\cdot 0\cdot r)) \;\middle|\; \begin{array}{l}(\rho_1,\delta,q)\in\mathcal{E}_{|\delta|-1} \wedge 0 < \delta \le p \wedge a \le v \le b \\ r \in \mathbb{R}^{p-2} \wedge \forall\, 0 \le n < p-2 : r_n = 0\end{array} \,\right\} \\
\cup & \left\{\, ((\rho_1,\delta,q),(\rho_1[V_1 \mapsto \nu],\delta-1,q)) \;\middle|\; (\rho_1,\delta,q)\in\mathcal{E}_p \wedge \delta > p \wedge a \le \nu \le b \,\right\}
\end{aligned}$$

Fig. 12. Abstract semantics of simple program P_1 with queues of length $p \geq 1$. The case of P_2 is similar.

input value, and adds it on top of the queue, discarding the value at the bottom at the queue.

2. If program P_k is behind the other program in the input sequence, and the delay is less than the size of the input queue, then a new successful input read operation retrieves the value in the queue indexed by this delay, and resets this value to zero.
3. If program P_k is behind the other program in the input sequence, and the delay is more than the size of the input queue, then a new successful input read operation produces a fresh input value, and leaves the queue unchanged.

In any case, an input read operation is only successful if the value read matches the bounds specified for the **input** statement. We do not display the semantics for other commands, as the semantics for assignments and tests are standard for memory environments, and leave input index differences and queues unchanged. For instance, $\hat{\mathbb{S}}^p_k[\![\, V \leftarrow e \,]\!] \triangleq \{\, ((\rho,\delta,q),(\rho[V \mapsto v],\delta,q)) \,|\, (\rho,\delta,q) \in \mathcal{E}_p \wedge v \in \mathbb{E}[\![\, e \,]\!]\rho \,\}$.

Double programs We then lift the semantics $\hat{\mathbb{S}}^p_1[\![\, s \,]\!]$ and $\hat{\mathbb{S}}^p_2[\![\, s \,]\!]$ to double programs. The definition of $\hat{\mathbb{D}}^p[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{D}}_p \times \hat{\mathcal{D}}_p)$ is very similar to that of $\hat{\mathbb{D}}[\![\, s \,]\!]$. The main change is that $\hat{\mathbb{D}}^p_k[\![\, s \,]\!]$ is defined with $\hat{\mathbb{S}}^p_k[\![\, s \,]\!]$, where $\hat{\mathbb{D}}_k[\![\, s \,]\!]$ is defined with $\hat{\mathbb{S}}_k[\![\, s \,]\!]$. We thus only show the definitions of some relations on Fig. 13. These definitions are very similar to those of $\hat{\mathbb{D}}[\![\, s \,]\!]$ on Fig. 10. The semantics for double statements and conditions compose the semantics of their left and right projections. The order of the composition is arbitrary, but significant for statements, as $\hat{\mathbb{D}}^p_1[\![\, s \,]\!] \fatsemi \hat{\mathbb{D}}^p_2[\![\, t \,]\!] \neq \hat{\mathbb{D}}^p_2[\![\, t \,]\!] \fatsemi \hat{\mathbb{D}}^p_1[\![\, s \,]\!]$. Both composition orders, however, are sound. A way to make the analyse precise and independent on the order would be to compute the intersection of the compositions with the two orders. The order is in constrast not significant for conditions, as $\hat{\mathbb{F}}^p_1[\![\, c \,]\!] \fatsemi \hat{\mathbb{F}}^p_2[\![\, d \,]\!] = \hat{\mathbb{F}}^p_2[\![\, d \,]\!] \fatsemi \hat{\mathbb{F}}^p_1[\![\, c \,]\!]$.

Finally, we formalize the relation between the abstract semantics $\hat{\mathbb{D}}^p$ and the previous abstraction $\hat{\mathbb{D}}$ of the concrete collecting semantics.

Proposition 4. *For all $p \geq 1$, $\hat{\mathbb{D}}^p$ is a sound and optimal abstraction of $\hat{\mathbb{D}}$:* $\hat{\mathbb{D}}^p = \alpha_p(\hat{\mathbb{D}})$.

$\hat{\mathbb{D}}^p[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{D}}_p \times \hat{\mathcal{D}}_p)$

$$\begin{aligned}
\hat{\mathbb{D}}^p[\![\, s_1 \parallel s_2 \,]\!] &\triangleq \hat{\mathbb{D}}^p_1[\![\, s_1 \,]\!] \fatsemi \hat{\mathbb{D}}^p_2[\![\, s_2 \,]\!] \\
\hat{\mathbb{D}}^p_1[\![\, s \,]\!] &\triangleq \{\, ((\rho_1, \rho_2, \delta, q), (\rho_1', \rho_2, \delta', q')) \mid ((\rho_1, \delta, q), (\rho_1', \delta', q')) \in \hat{\mathbb{S}}^p_1[\![\, s \,]\!] \wedge \rho_2 \in \mathcal{E} \,\} \\
\hat{\mathbb{D}}^p_2[\![\, s \,]\!] &\triangleq \{\, ((\rho_1, \rho_2, \delta, q), (\rho_1, \rho_2', \delta', q')) \mid ((\rho_2, \delta, q), (\rho_2', \delta', q')) \in \hat{\mathbb{S}}^p_2[\![\, s \,]\!] \wedge \rho_1 \in \mathcal{E} \,\} \\
\\
\hat{\mathbb{F}}^p[\![\, c_1 \parallel c_2 \,]\!] &\triangleq \hat{\mathbb{F}}^p_1[\![\, c_1 \,]\!] \fatsemi \hat{\mathbb{F}}^p_2[\![\, c_2 \,]\!] \\
\hat{\mathbb{F}}^p_k[\![\, c \,]\!] &\triangleq \{\, ((\rho_1, \rho_2, \delta, q), (\rho_1, \rho_2, \delta, q)) \mid (\rho_1, \rho_2, \delta, q) \in \hat{\mathcal{D}}_p \wedge \mathrm{true} \in \mathbb{C}[\![\, c \,]\!]\rho_k \,\} \;;\; k \in \{1; 2\}
\end{aligned}$$

Fig. 13. Abstract semantics of double programs with queues of length $p \geq 1$

2.3 Numerical abstraction

We now rely on numeric abstractions to abstract further $\hat{\mathbb{D}}^p[\![\, s \,]\!]$ into a computable abstract semantics $\hat{\mathbb{D}}^{\sharp p}[\![\, s \,]\!]$, resulting in an effective static analysis.

Connecting to numeric domains As $\hat{\mathcal{D}}_p \approx \mathbb{R}^{2|\mathcal{V}|+p+1}$, any numeric abstract domain with $2|\mathcal{V}|+p+1$ dimensions may be used, such as polyhedra [5]. Let $\mathcal{N}$ be such an abstract domain, with values in $\mathcal{D}^\sharp$, order $\sqsubseteq^\sharp$, concretization $\gamma_{\mathcal{N}} \in \mathcal{D}^\sharp \to \mathcal{P}(\mathbb{R}^{2|\mathcal{V}|+p+1})$, and operators $\hat{\mathbb{S}}^{\sharp p}[\![\, s \,]\!], \hat{\mathbb{C}}^{\sharp p}[\![\, c \,]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ for assignments and tests of simple programs over variables in $\mathcal{V}_1 \cup \mathcal{V}_2 \cup \mathcal{Q}$, where $\mathcal{V}_k \triangleq \{\, x_k \mid x \in \mathcal{V} \,\}$, and $\mathcal{Q} \triangleq \{\delta, (q_n)_{0 \leq n < p}\}$. Let $\cup^\sharp$ and $\cap^\sharp$ be the abstractions of set union and intersection of domain $\mathcal{N}$, and $\triangledown$ be its widening operator.

We abstract $\hat{\mathbb{D}}^p[\![\, s \,]\!] \in \mathcal{P}(\hat{\mathcal{D}}_p \times \hat{\mathcal{D}}_p)$ by $\hat{\mathbb{D}}^{\sharp p}[\![\, s \,]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$, with the soundness condition $\forall X^\sharp \in \mathcal{D}^\sharp : \hat{\mathbb{D}}^p[\![\, s \,]\!](\gamma_{\mathcal{N}}(X^\sharp)) \subseteq \gamma_{\mathcal{N}}(\hat{\mathbb{D}}^{\sharp p}[\![\, s \,]\!](X^\sharp))$. As $\hat{\mathbb{D}}^p[\![\, s \,]\!]$ is defined by induction on the syntax, the definition for $\hat{\mathbb{D}}^{\sharp p}[\![\, s \,]\!]$ is straightforward: the abstract semantics needs only be defined for the composition of syntactically different statements $s_1 \parallel s_2$ and conditions $c_1 \parallel c_2$. Fig. 14 shows definitions for associate transfer functions, as well as the transfer functions for some of the other syntactic constructs. We use the syntactic renaming operator τ_1 (resp. τ_2), defined by induction on the syntax, to distinguish the variables of the left (resp. right) projection of a double program, with suffix 1 (resp. 2). For instance, $\hat{\mathbb{D}}^{\sharp p}[\![\, c \leftarrow 1 \parallel 0 \,]\!] = \hat{\mathbb{S}}^{\sharp p}[\![\, c_2 \leftarrow 0 \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, c_1 \leftarrow 1 \,]\!]$.

Leveraging standard numeric domains Coming back to the example Unchloop from Fig. 3, recall that the relation between c and i is non linear: $c_1 = i_1 \times b_1 + 1$ and $c_2 = i_2 \times b_2$ from line 4 to line 9. Thus, a separate analysis of programs P_1 and P_2 would require a non linear abstract domain to compare r_1 and r_2. In contrast, our joint analysis of P_1 and P_2 will be sufficiently precise, even when using linear numeric domains, because the difference between the values of the variables in P_1 and in P_2 remains linear. For instance, the polyhedra domain [5] is able to infer that the invariant $-c_1 + c_2 + 1 = 0$ holds from line 3 to 9, hence $r_1 = r_2$ at line 9, although it is not able to discover any interval for r_1 or r_2. The octagon domain [18] is also able to express these invariants, but its transfer function for assignment is not precise enough to infer them. Indeed, $x \leftarrow a - b$

$$\hat{\mathbb{D}}^{\sharp p}[\![\, s \,]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$$

$$\hat{\mathbb{D}}^{\sharp p}[\![\, s_1 \parallel s_2 \,]\!] \triangleq \hat{\mathbb{D}}_2^{\sharp p}[\![\, s_2 \,]\!] \circ \hat{\mathbb{D}}_1^{\sharp p}[\![\, s_1 \,]\!]$$
$$\hat{\mathbb{D}}_k^{\sharp p}[\![\, s \,]\!] \triangleq \hat{\mathbb{S}}^{\sharp p}[\![\, \tau_k(s) \,]\!] \;;\; k \in \{1;2\}$$

$$\hat{\mathbb{F}}^{\sharp p}[\![\, c_1 \parallel c_2 \,]\!] \triangleq \hat{\mathbb{F}}_2^{\sharp p}[\![\, c_2 \,]\!] \circ \hat{\mathbb{F}}_1^{\sharp p}[\![\, c_1 \,]\!]$$
$$\hat{\mathbb{F}}_k^{\sharp p}[\![\, c \,]\!] \triangleq \hat{\mathbb{C}}^{\sharp p}[\![\, \tau_k(c) \,]\!] \;;\; k \in \{1;2\}$$

$$\hat{\mathbb{D}}^{\sharp p}[\![\, V \leftarrow e_1 \parallel e_2 \,]\!] \triangleq \hat{\mathbb{S}}^{\sharp p}[\![\, V_2 \leftarrow \tau_2(e_2) \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow \tau_1(e_1) \,]\!]$$

$$\hat{\mathbb{D}}_1^{\sharp p}[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] \triangleq \hat{\mathbb{S}}^{\sharp p}[\![\, \delta \leftarrow \delta - 1 \,]\!] \circ \left(\begin{array}{l} \hat{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow q_0 \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, q_0 \leftarrow \mathbf{rand}(a,b) \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, q_1 \leftarrow q_0 \,]\!] \circ \cdots \circ \hat{\mathbb{S}}^{\sharp p}[\![\, q_{p-1} \leftarrow q_{p-2} \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta \leq 0 \,]\!] \quad \cup^\sharp \\ \hat{\mathbb{S}}^{\sharp p}[\![\, q_{\delta-1} \leftarrow 0 \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow q_{\delta-1} \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, q_{\delta-1} \leq b \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, q_{\delta-1} \geq a \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta \leq p \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta > 0 \,]\!] \cup^\sharp \\ \hat{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow \mathbf{rand}(a,b) \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta > p \,]\!] \end{array} \right)$$

$$\hat{\mathbb{D}}_2^{\sharp p}[\![\, V \leftarrow \mathbf{input}(a,b) \,]\!] \triangleq \hat{\mathbb{S}}^{\sharp p}[\![\, \delta \leftarrow \delta + 1 \,]\!] \circ \left(\begin{array}{l} \hat{\mathbb{S}}^{\sharp p}[\![\, V_2 \leftarrow q_0 \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, q_0 \leftarrow \mathbf{rand}(a,b) \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, q_1 \leftarrow q_0 \,]\!] \circ \cdots \circ \hat{\mathbb{S}}^{\sharp p}[\![\, q_{p-1} \leftarrow q_{p-2} \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta \geq 0 \,]\!] \quad \cup^\sharp \\ \hat{\mathbb{S}}^{\sharp p}[\![\, q_{-\delta-1} \leftarrow 0 \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, V_2 \leftarrow q_{-\delta-1} \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, q_{-\delta-1} \leq b \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, q_{-\delta-1} \geq a \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta \geq -p \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta < 0 \,]\!] \cup^\sharp \\ \hat{\mathbb{S}}^{\sharp p}[\![\, V_2 \leftarrow \mathbf{rand}(a,b) \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta < -p \,]\!] \end{array} \right)$$

where $\tau_k(x) \triangleq \begin{cases} x_k & \mathbf{if}\ x \in \mathcal{V} \\ x & \mathbf{if}\ x \in \mathcal{Q} \end{cases}$

Fig. 14. Abstract semantics of double programs with a standard numeric domain

$$(\mathcal{P}(\hat{\mathcal{D}}_p \times \hat{\mathcal{D}}_p), \subseteq) \underset{\alpha_-}{\overset{\gamma_-}{\leftrightarrows}} (\mathcal{P}(\hat{\mathcal{D}}_p \times \hat{\mathcal{D}}_p), \subseteq)$$

$$\alpha_-(R) \triangleq \left\{ ((\rho_1, \rho_2 - \rho_1, \delta^\star, q), (\rho_1', \rho_2' - \rho_1', \delta^{\star\prime}, q')) \;\middle|\; ((\rho_1, \rho_2, \delta^\star, q), (\rho_1', \rho_2', \delta^{\star\prime}, q')) \in R \right\}$$

$$\gamma_-(\Delta) \triangleq \left\{ ((\rho_1, \rho_1 + \delta_\rho, \delta^\star, q), (\rho_1', \rho_1' + \delta_\rho', \delta^{\star\prime}, q')) \;\middle|\; ((\rho_1, \delta_\rho, \delta^\star, q), (\rho_1', \delta_\rho', \delta^{\star\prime}, q')) \in \Delta \right\}$$

Fig. 15. Abstraction of double environments with environment differences

cannot be exactly abstracted by the domain, and currently proposed transfer functions fall back to plain interval arithmetics in that case, so that the domain cannot exploit the bound it infers on $a - b$ to bound x, for efficiency reasons. The interval domain is not able to express the invariants, hence it cannot be used directly for a conclusive analysis.

2.4 Introducing a dedicated numeric domain

However, we remark that it is sufficient to bound the difference $x_2 - x_1$ for any variable x to express the necessary invariants, where x_1 (resp. x_2) represents the value of x for the left (resp. right) projection P_1 (resp. P_2) of a double program P. Thus, we now design an abstract domain that is specialized to infer these bounds. We abstract the values x_1 and x_2 by the pair $(x_1, \delta x)$, where $\delta x \triangleq x_2 - x_1$. This abstraction amounts to changing the representation of states of double program P. It does not lose information. A formalization of this abstraction is shown on Fig. 15. Note that we extend operators $+$ and $-$ to functions (pointwise lifting).

Proposition 5. *The pair (α_-, γ_-) defined in Fig. 15 is a Galois isomorphism.*

Let $\mathbb{\Delta}^p \triangleq \alpha_-(\hat{\mathbb{D}}^p)$. $\mathbb{\Delta}^p$ is able to represent two-variable equalities $x_1 = x_2 \Leftrightarrow \delta x = 0$, even after numeric abstraction using non relational domains, such as

$$
\begin{aligned}
&\mathbb{\Delta}^p[\![\, c \leftarrow c + b]\!] \\
&\quad = \{\, (s_1, (((a_1, b_1, c_1 + b_1, i_1, r_1), ((a_1 + \delta a) - a_1, (b_1 + \delta b) - b_1, \\
&\qquad\qquad ((c_1 + \delta c) + (b_1 + \delta b)) - (c_1 + b_1), (i_1 + \delta i) - i_1, (r_1 + \delta r) - r_1), \delta, q)) \mid H_1 \,\} \\
&\quad = \{\, (s_1, ((a_1, b_1, c_1 + b_1, i_1, r_1), (\delta a, \delta b, \delta c + \delta b, \delta i, \delta r), \delta, q)) \mid H_1 \,\} \\
&\mathbb{\Delta}^p[\![\, r \leftarrow c \parallel c + 1]\!] \\
&\quad = \{\, (s_1, ((a_1, b_1, c_1, i_1, c_1), ((a_1 + \delta a) - a_1, (b_1 + \delta b) - b_1, ((c_1 + \delta c) - c_1, \\
&\qquad\qquad (i_1 + \delta i) - i_1, (c_1 + \delta c + 1) - c_1)), \delta, q) \mid H_1 \,\} \\
&\quad = \{\, (s_1, ((a_1, b_1, c_1, i_1, c_1), (\delta a, \delta b, \delta c, \delta i, \delta c + 1), \delta, q)) \mid H_1 \,\} \\
&\text{where } s_1 \triangleq ((a_1, b_1, c_1, i_1, r_1), (\delta a, \delta b, \delta c, \delta i, \delta r), \delta, q) \\
&\text{and} \quad H_1 \triangleq s_1 \in \mathbb{R}^{10} \times \mathbb{Z} \times \mathbb{R}^p
\end{aligned}
$$

Fig. 16. Examples of $\mathbb{\Delta}^p$ semantics

$$
\begin{aligned}
&\mathbb{\Delta}^{\sharp p}[\![\, V \leftarrow \mathbf{input}(a, b)]\!] \triangleq \\
&\quad \Big(\hat{\mathbb{S}}^{\sharp p}[\![\, \delta V \leftarrow 0 \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow q_0 \,]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, q_0 \leftarrow \mathbf{rand}(a, b) \,]\!] \circ_{i=0}^{p-2} \hat{\mathbb{S}}^{\sharp p}[\![\, q_{i+1} \leftarrow q_i \,]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta^\star = 0 \,]\!] \cup^\sharp \\
&\qquad\qquad \mathbb{\Delta}_2^{\sharp p}[\![\, V \leftarrow \mathbf{input}(a, b)]\!] \circ \mathbb{\Delta}_1^{\sharp p}[\![\, V \leftarrow \mathbf{input}(a, b)]\!] \circ \hat{\mathbb{C}}^{\sharp p}[\![\, \delta^\star \neq 0 \,]\!] \Big) \\
&\mathbb{\Delta}^{\sharp p}[\![\, V \leftarrow e]\!] \triangleq \mathbb{\Delta}_2^{\sharp p}[\![\, V \leftarrow e]\!] \circ \hat{\mathbb{S}}^{\sharp p}[\![\, V_1 \leftarrow (\tau_1 \circ \pi_1)(e) \,]\!] \\
&\text{where} \\
&\mathbb{\Delta}_2^{\sharp p}[\![\, V \leftarrow e]\!] \triangleq \begin{cases} \hat{\mathbb{S}}^{\sharp p}[\![\, \delta V \leftarrow 0 \,]\!] & \textbf{if } \text{is\_deterministic}(e) \wedge \forall x \in \text{Vars}(e) : \delta x = 0 \\ \hat{\mathbb{S}}^{\sharp p}[\![\, \delta V \leftarrow \sum_{x \in \mathcal{V}} \lambda_x \delta x \,]\!] & \textbf{if } \exists (\mu, (\lambda_x)_{x \in \mathcal{V}}) \in \mathbb{R}^{|\mathcal{V}|+1} : e = \mu + \sum_{x \in \mathcal{V}} \lambda_x x \\ \hat{\mathbb{S}}^{\sharp p}[\![\, \delta V \leftarrow (\tau_2' \circ \pi_2)(e) - (\tau_1 \circ \pi_1)(e) \,]\!] & \textbf{otherwise} \end{cases} \\
&\tau_2'(x) \triangleq \begin{cases} x_1 + \delta x & \textbf{if } x \in \mathcal{V} \\ x & \textbf{if } x \in \mathcal{Q} \end{cases}
\end{aligned}
$$

Fig. 17. Symbolic simplifications in $\mathbb{\Delta}^{\sharp p}$

intervals. Transfer functions rely on symbolic simplifications to let such equalities propagate through linear expressions. The semantics $\mathbb{\Delta}^p$ of statements 6 and 9 of the UnchLoop example are shown for instance on Fig. 16, before and after simple symbolic simplifications of affine expressions.

Like for $\hat{\mathbb{D}}^p$, any numeric domain over variables in $\mathcal{V}_1 \cup \mathcal{V}_\delta \cup \mathcal{Q}$, where $\mathcal{V}_\delta \triangleq \{\, \delta x \mid x \in \mathcal{V} \,\}$, can be used to abstract $\mathbb{\Delta}^p$. Therefore the definition for $\mathbb{\Delta}^{\sharp p}$ is straightforward, by induction on the syntax of double programs. We also define the semantics for the $s_1 \parallel s_2$ construct as $\mathbb{\Delta}^{\sharp p}[\![\, s_1 \parallel s_2]\!] \triangleq \mathbb{\Delta}_2^{\sharp p}[\![\, s_2]\!] \circ \mathbb{\Delta}_1^{\sharp p}[\![\, s_1]\!]$, where $\mathbb{\Delta}_1^p[\![\, s]\!] \triangleq \mathbb{\Delta}^p[\![\, s \parallel \mathbf{skip}]\!]$, and $\mathbb{\Delta}_2^p[\![\, s]\!] \triangleq \mathbb{\Delta}^p[\![\, \mathbf{skip} \parallel s]\!]$, for simple statement s. Nonetheless, we add some particular cases, to gain both efficiency and precision on δV, for all variables V, through simple symbolic simplifications. These particular cases are displayed on Fig. 17. Note that we use the syntactic renaming operator τ_2', defined by induction on the syntax, to replace variables V_2 of the right projection of a double program by their abstraction $V_1 + \delta V$.

The first particular case is that of input statements $V \leftarrow \mathbf{input}(a, b)$ for both programs, in environments such that both programs have read the same number of input values, *i.e.* $\delta^\star = 0$, where $\delta^\star$ represents the difference between input indexes. In this case, we may assign $\delta V \leftarrow 0$ directly, and leave $\delta^\star$ unchanged.

For instance, after statement $a \leftarrow \mathbf{input}(-1000, 1000)$ at line 1 of the Unchloop example on Fig. 3, we have $a \in [-1000, 1000]$, and $\delta a = 0$. The second particular case is that of affine assignments $V \leftarrow e$, where $e = \mu + \sum_{x \in \mathcal{V}} \lambda_x \times x$. We call such expressions "differentiable", as it is easy to compute δV directly as a function of all the δx variables. A third particular case is that of arbitrary (non necessarily affine) assignments $V \leftarrow e$, when e is deterministic, and all the occurring variables x satisfy $\delta x = 0$. Then $\delta V = 0$, as we know that both expressions always evaluate to equal values in P_1 and P_2.

To further enhance precision on some examples, we slightly generalize these particular cases to double assignments $V \leftarrow e_1 \parallel e_2$, when expressions e_1 and e_2 are found syntactically equal, modulo some semantics preserving transformations, such as associativity, commutativity, and distributivity. We also generalize symbolic simplifications based on expression differentation to some double assignments $V \leftarrow e \parallel e + e'$, in particular when e' is a constant. For instance, for line 9 of the Unchloop example on Fig. 3, we have $\triangle_2^{\sharp p}[\![r \leftarrow c \parallel c + 1]\!] = \hat{\mathbb{S}}^{\sharp p}[\![\delta r \leftarrow \delta c + 1]\!]$.

As a consequence, the interval domain is able to infer the invariant $\delta r = 0$ for semantics $\triangle^{\sharp p}$ at line 10 of this example, resulting a conclusive analysis with linear cost, which is much more efficient than using polyhedra with $\hat{\mathbb{D}}^{\sharp p}$.

3 Evaluation

We implemented a prototype abstract interpreter for the semantics $\hat{\mathbb{D}}^{\sharp p}$ and $\triangle^{\sharp p}$ of the toy language introduced in this paper. It is about 2,500 lines of OCaml source code. It uses the Apron [12] library to experiment with the polyhedra and octagon abstract domains, and the BddApron [13] library to implement state partitioning.

3.1 Benchmarking

We compare results on small examples selected from other authors' benchmarks [24,21,22]. Note that some of these benchmarks originate from real patches in GNU core utilities. We added a larger benchmark (also from a Coreutils patch), to evaluate scalability. For most benchmarks, patches preserve most of the loop and branching structure, except for the seq benchmark from [21,22], which features deep modifications of the control structure. The related works do not address streams. As a consequence, these benchmarks do not feature unbounded reads into input streams, except for the remove benchmark, which we presented in the introduction: see Fig. 1 and Fig. 2. Note that we simplified this benchmark to fstatat caching for a single file, in order to compare with [21].

[24,21,22] deal with C programs directly, while we encode their benchmarks in our toy language. In addition, these references not only prove equivalences, but also characterize differences, while we focus on equivalence for now. We therefore selected benchmarks relevant to equivalence only, except for the [24,

Fig. 2] example, which we modified slightly to restore equivalence of terminating executions: see Fig.4. On the other hand, [24] gives several versions of their benchmarks, depending on the maximum numbers of loop iterations of the examples. Indeed, the symbolic execution technique they use is very sensitive to this parameter. We do not have this constraint, as we use widening instead of fully unrolling loops, so that we handle directly unbounded loops in a sound way.

Figure 18 summarises the results of our analysis. It shows the analysis timings and results of our prototype, as well as timings of the analyses the related work, when they are available (their analyses are all successful). All experiments were conducted on a Intel® Core-i7™ processor. Our results are comparable with those of the original authors, with speedups of one order of magnitude or more. Some timing differences, of the order of milliseconds, cannot be considered significant, especially as the experiments are not performed on the same machines. A significant point, however, is that the benchmark LoopMult takes 49 seconds in [24], which is two orders of magnitude slower than the benchmark Const, while, with our method, both Const and LoopMult are analyzed at roughly the same speed. This difference in behaviors can be explained as a benefit of widening over unrolling loops. Hence, our timing comparison proves that our method can achieve at worst a similar speed, and it is also much more scalable for problems difficult in previous work. Note that [24] compared their method to well-established tools, such as Symdiff [14] and RVT [8], and observed speedups of one order of magnitude and more with respect to them. Therefore, it is not useful to compare our prototype with these tools on these benchmarks.

Most benchmarks are analyzed successfully with the polyhedra domain, without partitioning. The seq benchmark, for instance, is analyzed precisely despite significant changes in the control structure, as the matching of statements is established as part of the syntax of double programs. Only the remove benchmark requires partitioning for a successful analysis with the polyhedra domain. Four other benchmarks are analyzed very efficiently with the non relational interval domain, thanks to the $\mathbb{\Delta}^{\sharp p}$ semantics. Partitioning improves the precision on three other, but reduces efficiency. Nonetheless, some benchmarks, such as LoopSub, cannot be analysed conclusively using a non relational numeric domain with semantics $\hat{\mathbb{D}}^{\sharp p}$ or $\mathbb{\Delta}^{\sharp p}$. Indeed, related patches exchange the roles of two variables a and b, so that the challenge is not to infer $a_1 = a_2 \wedge b_1 = b_2$, but $a_1 = b_2 \wedge b_1 = a_2$. We therefore developed a dedicated abstract domain, to refine $\hat{\mathbb{D}}^{\sharp p}$ with automatically inferred variable equalities. This domain is based on union-find data structure that maintains a partitioning of the set $\mathcal{V}_1 \cup \mathcal{V}_2 \cup \mathcal{Q}$ of program variables. Two variables are part of the same equivalence class if they are guaranteed to be equal. The associate abstract lattice is the dual of the standard geometric lattice of partitions of a finite set: $a \sqsubseteq b$ means that partition b refines partition a, *i.e.* every equivalence class of a is a union of classes of b; $\top$ is the set of singleton variables; and the smallest non $\bot$ element is the whole set of variables. This abstract lattice has finite height, so we use union in place of widening. The LoopSub benchmark is analysed successfully using a reduced product between intervals and this domain.

Related origin	Benchmark	LOC	Related time	$\mathbb{D}^{\sharp 1}$(polyhedra) Partitioning No	$\mathbb{D}^{\sharp 1}$(polyhedra) Partitioning Yes	$\mathbb{D}^{\sharp 1}$(octagon) Partitioning No	$\mathbb{D}^{\sharp 1}$(octagon) Partitioning Yes	$\mathbb{A}^{\sharp 1}$(interval) Partitioning No	$\mathbb{A}^{\sharp 1}$(interval) Partitioning Yes
[24]	Comp	13	539 ms	14 ms ✓		18 ms ✗		2 ms ✗	
	Const	9	541 ms	7 ms ✓		17 ms ✓		1 ms ✓	
	Fig. 2	14	–	4 ms ✓		5 ms ✓		1 ms ✓	
	LoopMult	14	49[2] s	20 ms ✓		56 ms ✗		1 ms ✗	
	LoopSub	15	1.2 s	19 ms ✓		53 ms ✗		2 ms ✗	
	UnchLoop	13	2.8[3] s	15 ms ✓		36 ms ✗		2 ms ✓	
[21]	sign	12	–	6 ms ✓		8 ms ✗	420 ms ✓	2 ms ✗	400 ms ✓
	sum	14	4 s	14 ms ✓		30 ms ✓		6 ms ✗	3.2 s ✓[4]
	copy[1]	37	7 s	102 ms ✓		60 ms ✓		2 ms ✗	430 ms ✓
	remove[1]	19	1 s	31.6 s ✗	481 ms ✓	42 ms ✗	322 ms ✓	7 ms ✗	
[21,22]	seq[1]	41	11 s	75 ms ✓		500 ms ✗		2 ms ✗	
	test[1]	158	–	96 ms ✓		521 ms ✓		4 ms ✓	

Fig. 18. Benchmarks

3.2 Handling streams

All benchmarks of table 18 were analyzed using fixed-length queues of length 1, as the related works do not handle input streams. Note that abstracting infinite input streams with fixed-length queues of length 1 is also enough to analyze some patches of infinite-state programs with unbounded loops reading from a stream (e.g. a file), even when patches reorder input statements across the body of unbounded loops.

Fig.19 shows an example. This patch reorders input statements in the loop, and changes the number of input statements in terminating executions. The loop is unbounded, and the program is infinite-state. Terminating executions of the left and right projections compute equal values for s, though possibly not for x. This double program is analyzed successfully with $\mathbb{D}^{\sharp 1}$, using any relational numerical domain: 33 ms for polyhedra, 43 ms for octagon, and 18 ms for the reduced product between the domains of intervals and variable equalities. To the best of our knowledge, no previous work has sound and precise automatic analyses for patches of this type.

In the bounded abstraction of streams, the unbounded FIFO queue represents the subsequence of input values read by the program ahead in the sequence, and not yet read by the program behind. Though we are bounding this queue in the abstract, we retain precise information on executions reading arbitrary long input sequences. The bounded queue allows retaining relational information between all input values read with delays less or equal to the bound, while non relational (interval) information is retained for values read with larger delays. Fig. 20 shows a simple example. Using a queue of length 1 is enough to infer the range of variable s in both projections of the double program. On the contrary,

[1] Coreutils [2]only 20 loop iterations [3]only 5 loop iterations [4]only 32 values of len

```
1   s = input(−5,5);
2   b = input(0,1);
3   { x = input(0,10); } ‖ {/* skip */}
4   while ( b == 1 ) {
5     {/* skip */} ‖ { x = input(0,10); }
6     s = s + x;
7     b = input(0,1);
8     { x = input(0,10); } ‖ {/* skip */}
9   }
10  assert_sync(s);
```

Fig. 19. Reordering reads from an input stream

```
1   { a = input(0,5); a = input(−5,0); } ‖ {/* skip */}
2   x = input(0,5);
3   x = input(−5,0);
4   s = a ‖ x;
5   assert(−5 <= s && s <= 0); // inferred with with a queue of length p ≥ 1
6   assert_sync(s); // inferred with a queue of length p ≥ 2
```

Fig. 20. Relational and non relational information versus lengths of queues

a queue of length at least 2 is necessary to prove that both programs compute equal values for s.

4 Conclusion

We presented a static analysis for software patches. Our method is based on abstract interpretation, and parametric in the choice of an abstract domain. We presented a novel concrete collecting semantics, expressing the behaviors of two syntactically close versions of a program at the same time. This semantics deals with programs reading from unbounded input streams. We also introduced a novel numeric domain to bound differences between the values of the variables in the two programs, which has linear cost. We implemented a prototype and experimented on a few small examples from the literature.

In future work, we will consider extensions to larger, and non purely numeric programs, towards the analysis of realistic patches. We will also extend our method to characterize the semantic differences between two non equivalent versions of program. We will also investigate to what extend our approach could generalize to portability analysis, a dual problem where we wish to compare the semantics of the same program in two different environments. We plan to experiment with other abstract domains for our analysis, such as zonotopes. Finally, we will investigate the connections between our semantics and information flow problems. Indeed, as a side-effect of our method, our analysis is able to prove

```
1  pub = input(−10,10);
2  sec = rand(−5,5);
3  if (sec < 0) pub = 1;
4  pub = 0;
5  assert_sync(pub); // OK
```

(a) secure program

```
1  pub = input(−10,10);
2  sec = rand(−5,5);
3  if (sec < 0) pub = 1;
4  pub = pub + 1;
5  assert_sync(pub); // failed
```

(b) insecure program

Fig. 21. Proving information flow properties

that two sets of executions of the same program compute equal values for some outputs. This is useful for proving some information flow properties, such as secrecy. For instance, Fig. 21 shows two programs with public variable pub and secret variable sec. These programs read pub as an input value, and choose sec non-deterministically, For all pairs of executions reading equal values in pub, but possibly different values in sec, Program 21(a) computes equal values for pub. hence ensuring secrecy. On the contrary, Program 21(b) leaks the sign of sec. Our analysis is able to distinguish these two programs. Indeed, it compares the semantics of two versions of each program. In this case, both versions have exactly the same code, which is a form of self-composition [3,20].

References

1. The Be Book, https://www.haiku-os.org/legacy-docs/bebook/index.html
2. The Haiku Operating System, https://www.haiku-os.org/
3. Barthe, G., D'argenio, P.R., Rezk, T.: Secure information flow by self-composition. Mathematical. Structures in Comp. Sci. **21**(6), 1207–1252 (Dec 2011)
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77. pp. 238–252. ACM (Jan 1977)
5. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL'78. pp. 84–97. ACM (1978)
6. Delmas, D., Miné, A.: Analysis of Program Differences with Numerical Abstract Interpretation. In: PERR 2019. Prague, Czech Republic (Apr 2019)
7. Floyd, R.W.: Assigning meanings to programs. Proceedings of Symposium on Applied Mathematics **19**, 19–32 (1967)
8. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of DAC '09. pp. 466–471. ACM, New York, NY, USA (2009)
9. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI. pp. 232–247 (2011)
10. Goubault, E., Putot, S.: Robustness analysis of finite precision implementations. In: Programming Languages and Systems. pp. 50–57 (2013)
11. Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In: Proceedings of ICSM '94. pp. 243–252 (1994)
12. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Proc. of CAV'09. vol. 5643, pp. 661–667 (June 2009)
13. Jeannet, B.: Bddapron: A logico-numerical abstract domain library (2009), `http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/`

14. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: CAV. pp. 712–717 (2012)
15. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Proceedings of ESEC/FSE 2013. pp. 345–355 (2013)
16. Marinescu, P.D., Cadar, C.: Katch: High-coverage testing of software patches. In: Proceedings of ESEC/FSE 2013. pp. 235–245 (2013)
17. Martel, M.: Propagation of roundoff errors in finite precision computations: A semantics approach. In: Programming Languages and Systems. pp. 194–208 (2002)
18. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**(1), 31–100 (2006)
19. Mora, F., Li, Y., Rubin, J., Chechik, M.: Client-specific equivalence checking. In: Proceedings of ASE 2018. pp. 441–451 (2018)
20. Müller, C., Kovács, M., Seidl, H.: An analysis of universal information flow based on self-composition. In: CSF 2015. pp. 380–393 (July 2015)
21. Partush, N., Yahav, E.: Abstract semantic differencing for numerical programs. In: SAS. pp. 238–258 (2013)
22. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: Proceedings of OOPSLA'14. pp. 811–828 (2014)
23. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT'08/FSE-16. pp. 226–237 (2008)
24. Trostanetski, A., Grumberg, O., Kroening, D.: Modular demand-driven analysis of semantic difference for program versions. In: Proceedings of SAS 2017. pp. 405–427 (2017)