Computation over partial information

Ian Sabourin
University of Montreal
Montreal, Canada
sabouian@iro.umontreal.ca

Stefan Monnier
University of Montreal
Montreal, Canada
monnier@iro.umontreal.ca

Marc Feeley
University of Montreal
Montreal, Canada
feeley@iro.umontreal.ca

ABSTRACT

We introduce *partial interpretation*, an approach to program execution over partial information. Partial interpretation aims to combine the accuracy of symbolic execution and the termination of abstract interpretation, through what we call *lossless abstraction*: discarding information which is subsequently irrelevant.

We distinguish several notions of partial information: sets, histories, and relations. We identify one lossless abstraction: under suitable conditions, discarding all history is lossless. We state this fact precisely and we prove it. In particular, in our object language (a mini-ML without closures), it is lossless to discard the history of the arguments to a call. Discarding this history is a step towards the principled accuracy and termination we seek.

We sketch a termination argument for partial interpretation. To ensure termination, further information must be discarded, and we do so in an unprincipled way. We then apply our approach to partial evaluation, reproducing a classic result without manual adjustment of the input program. On a separate synthetic example, accuracy is improved over previous partial evaluators. The performance of partial interpretation is initially poor, but we remedy this with *ad hoc* techniques.

CCS CONCEPTS

• **Theory of computation → Program analysis**; *Automated reasoning*; *Abstraction*; • **Software and its engineering** → Retargetable compilers.

KEYWORDS

Program analysis, Symbolic execution, Abstract interpretation, Partial evaluation

ACM Reference Format:
Ian Sabourin, Stefan Monnier, and Marc Feeley. 2021. Computation over partial information. In *Proceedings of Implementation and Application of Functional Languages (IFL'2021).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We are used to the following picture of an executing program: an input is provided, the program runs for a while, and a result comes out. We tacitly assume complete information about the input, the result, and any intermediate results in between.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
IFL'2021, Sep 2021, online
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this work, we ask what it would mean to execute a program over *partial information*, in a principled way. There exist approaches in this direction. Symbolic execution is accurate, but it has a termination problem. Abstract interpretation terminates, but it is approximate.

We combine these approaches through *lossless abstraction*. Starting from the complete accuracy of symbolic execution, we identify information which can be discarded because it is subsequently irrelevant. Discarding irrelevant information enables abstract interpretation techniques for termination, without an arbitrary choice of abstract domain (i.e. which information to keep). We refer to our combined approach as *partial interpretation*.

We distinguish several notions of partial information: sets, histories, and relations. We identify one lossless abstraction which motivates partial interpretation: discarding the history of the arguments to every call. To ensure termination, further information must be discarded. We do so in an unprincipled way, but we feel that our work is a step towards a more principled solution, with lossless abstraction as a general idea.

The main piece of our presentation is our lossless abstraction, a demonstration that it is lossless, and a description of partial interpretation. This may be relevant to accurate and terminating program analysis, in general.

In addition, our work especially relates to *partial evaluation*: automatic program specialization. Partial evaluation is reformulated so that its main sub-problem is computation over partial information. We argue that a principled approach is especially relevant here. We re-use ideas from past work on online partial evaluation, in the unprincipled part of our approach. Finally, we validate our approach by applying it to partial evaluation.

We make the following contributions:

- a distinction between several notions of partial information: sets, histories, and relations
- a sufficient condition under which discarding all history is lossless, in particular for the arguments to a call
- a relational semantics for the variant of symbolic execution we use in partial interpretation
- definitions of discarding history, losslessness, and irrelevance in terms of our relational semantics
- a proof of irrelevance under our sufficient condition, and sanity checks on our definition of irrelevance
- a description of partial interpretation, and a sketch of its termination argument
- a formulation of partial evaluation in terms of partial interpretation
- an implementation of such a partial evaluator, with preliminary experimental results.

We organize the remaining sections as numbered here:

(2) we describe our object programs
(3) we discuss symbolic execution over them
(4) we introduce our lossless abstraction, and the architecture of partial interpretation
(5) we introduce our relational semantics for the symbolic execution we use in partial interpretation, we state and prove our main irrelevance theorem, and we check the sanity of its statement
(6) we study the termination of partial interpretation
(7) we apply our approach to partial evaluation
(8) we discuss related work
(9) we wrap up and discuss future work.

2 OBJECT PROGRAMS

We create our own notion of object programs, to adapt it to the needs of partial interpretation. Some decisions are relevant to partial interpretation; other decisions just keep things simple. The result is similar to a subset of ML.

We focus on pure functional programs, which we view as a well-understood programming core: a call-return discipline (relevant in Sec. 4), without mutation or external effects (to simplify). Our programs have standard static polymorphic types. We are about to do symbolic execution, we use an SMT solver for this, and it likes static types for all data.

We represent a (sub-)program as a graph similar to a compiler IR (intermediate representation), as in Fig. 1. Our graph representation - a *pgraph* - is relevant because our main irrelevance theorem involves a graph condition. Nodes are operations or constants, and edges are data dependencies: $A \rightarrow B$ indicates that A reads (the result of) B. If A has several out-edges, they are ordered and implicitly labelled.

We have eight node types, some with an annotation: *entry*; *tuple*; *proj* `i`; *const* `val`; *prim* `op`; *prog* `i`; *call*; *if*.

entry expresses a program's single formal parameter. A program returns a single result, expressed by specially marking a single node as the "exit" (here, the *if*).

tuple packs any number of data, and *proj* `i` extracts a component from a tuple. These account for "multiple" arguments or results. A nullary *tuple* is the unit value. A unary *tuple* is a no-op - intuitively, there are no one-element tuples.

const `val` and *prim* `op` express constants and primitive operations. Visually, only the `val` or `op` is shown. We use these to introduce booleans, the mathematical integers $\mathbb{Z}$, and operations over both. This directly matches SMT primitives.

prog `i` expresses as data the `i`th sub-program in an ambient context of *closed, top-level* programs. *We have no closures*: all input to a sub-program goes through its single argument. This is directly relevant to our lossless abstraction. We can imagine that any closures have already undergone closure conversion or defunctionalization. *call*'s first edge is to a program; its second edge is to the single argument. A single argument simplifies our presentation.

In principle, we should complete this with programmer-defined algebraic sums, e.g.: *constructor* `i`; *accessor* `i`; *case*. To simplify, we introduce only an *if* node, and primitives for homogeneously typed lists: `nil`, `cons`, `head`, `tail`.

if expresses a conditional. Its first edge is to a boolean condition c. Two additional edges (the *consequent* and *alternate* branches) indicate which data to select between, based on the value of c. *if* selects not just between values, but between *execution paths*: the condition is computed first, then only one branch is computed. Everywhere else, execution is *strict*.

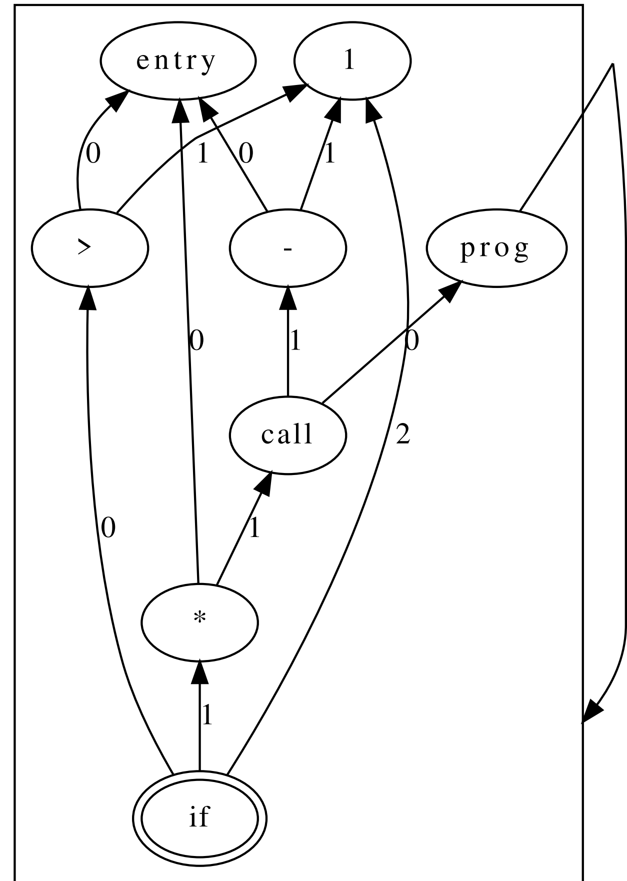

Figure 1: The *pgraph* of a factorial program.

3 SYMBOLIC EXECUTION

In an ordinary interpreter, we would map each *pgraph* node to an ordinary value. Symbolic execution is analogous: we map each node to a symbolic expression in the SMT-LIB language [Barrett et al. 2010]. We define this mapping by structural induction. We describe the behavior at each node type, and simultaneously we discuss the example of `fact(x)`: factorial of a completely unknown integer `x`.

At *entry*, our caller provides a symbolic input (here, `x`) accompanied by declarations for any variables, and propositions which constrain these variables (here, `x` is unconstrained).

const and *prog* produce simple expressions. We define an SMT type for programs, with a single constructor which wraps the integer identifier of the (sub-)program.

prim builds a symbolic expression - here `x>1` and `x-1`, and a product at `*` if the *call* terminates.

tuple and *proj* (not used here) are handled with a single-constructor SMT type for each tuple arity. We have implemented only pairs, but general tuples simplify our presentation.

if attempts to decide the condition by querying the SMT solver. We get a *model* (a possible value) for the condition under the current constraints. Then, the condition is temporarily asserted to be the opposite of the model, and SATisfiability is checked again. We get either one or two possible conditions.

When the condition is uniquely known, *if* directly reports the result of the corresponding branch. Otherwise (as here), *if* builds a conditional expression, written (`ite` c a b) - `ite` stands for if-then-else. It is possible to track a *path condition* here: compute the consequent branch *under the assumption that the condition is true* - and conversely in the alternate branch. We ignore path conditions in our presentation.

The simplest handling of *call* invokes symbolic execution recursively (here, with the argument `x-1`). Any relevant variables and propositions are passed on. We ignore the general question of which sub-program to call, which amounts to enumerating models for the called program. We focus on the case when the called program is uniquely known; if it is not, we report a completely unknown result from the *call*.

As described, symbolic execution does not terminate on recursive object programs. In our `fact(x)` example, the *if* cannot decide the condition, so it computes both branches before packaging them in a conditional expression. In the recursive branch, we compute `fact(x-1)` by invoking symbolic execution recursively. `x-1` is still completely unknown, and we are in an infinite loop. Tracking path conditions would change some details, but it would not get us termination.

This illustrates a fundamental problem: even when a program terminates on all completely known inputs, it does not necessarily terminate on partially known inputs. Partial interpretation addresses this by handling calls differently.

4 PARTIAL INTERPRETATION

We start from a key observation in the example of `fact(x)`: in the recursive call, `x-1` is *still completely unknown*. In some sense, there is repetition between these calls. Intuitively, the successive arguments are obviously distinct: the second is one less than the first. But taken as *sets*, the possible arguments to each call are the same: the set of all integers.

Our central intuition is that if the arguments are equal as sets, then the results are equal in some (lossless) sense, and we can leverage this to achieve termination. Our task now is to clarify in what sense the results are equal, to explain how we leverage this, and to prove that all this works.

We first distinguish many notions of (partial) information. In our introduction we mentioned complete information: a uniquely known value. We've just introduced sets, and complete information becomes a special case: a singleton set.

Now consider this. If the arguments `x` and `x-1` above are equal as sets, but they were distinct intuitively, *what were they distinct as*? There must be a third notion of partial information. We propose to think of it as a *computational history*: a log of how a piece of data has been produced.

With complete information, we are not used to caring about computational history. We report the result of $2 + 3$ as 5 without thinking twice. But reporting 5 *discards information*: the historical knowledge that 5 was produced as $2 + 3$, as opposed to e.g. $1 + 4$. In some sense we lose nothing from this, because the discarded information is never relevant in the future: anything true about $2 + 3$ is also true about 5, and vice-versa. Furthermore, we save resources by storing 5, which justifies it as "the result".[1]

With partial information, history can matter. Suppose that from a completely unknown integer x, we compute both $2x+1$ and $2x-1$, then subtract them. Intuitively we know that the difference is 2. However, consider what happens if we view $2x + 1$ and $2x - 1$ as sets, through the lens of mathematical set comprehension. $\{2x + 1 \mid x \in \mathbb{Z}\}$ and $\{2x - 1 \mid x \in \mathbb{Z}\}$ are equal: the set of odd integers. And the difference of any two odd integers is any even integer; not the singleton set $\{2\}$. Viewing $2x + 1$ and $2x - 1$ as sets discards their history: intuitively, the knowledge that they were computed from *the same* x. As a result, we lose accuracy in their difference.

In general, the most assuredly accurate form of partial information must be *complete computational history*, which is captured by the expressions of symbolic execution.

We now clarify in what sense the results are equal, when the arguments are equal as sets. Intuitively, we plan to share a single result across all calls with the same set of arguments. Sharing complete computational history (from the start of the program) would not do, since these various calls have different histories *prior to the call* - that is the whole point. We want a fourth and last notion of partial information: a history *from the start of the call*, which we can combine with any caller's prior history. Intuitively, we want an input-output relation. Concretely, we introduce a fresh variable for the formal parameter, when we begin symbolic execution of a call. Later, any caller replaces the formal parameter with the actual argument, in the result.

We re-iterate: if the arguments are equal as sets, then the results are equal as input-output relations. We now suppose this is correct, and explain how we leverage it. In the next section, we prove it.

`fact(x)` uses the result of `fact(x-1)` to compute its own result. But by supposition, these results are one and the same. Therefore that result depends on itself:

$$R = F(R)$$

where R is the result of `fact(x)`, and F represents part of the *pgraph* of `fact`: from the recursive call to the result. Restating this equation in words, R is a fixed point of F. We plan to compute this fixed point iteratively.

In general, we have one result for each pair `(pgraph, args)`, where the `args` are treated as sets. These results depend on one another, in a way determined by the object program. We seek a fixed point over all these results.

We introduce a global cache which maps a pair `(pgraph, args)` to a result - we discuss result initialization shortly. We then compute a fixed point over the entire cache. We re-compute a result by symbolically executing that `pgraph` for those `args` (as in Sec. 3), with one modification: at a *call*, we do not invoke symbolic execution recursively; instead, we read a result from the cache.

[1] However, debugging might be easier if we knew it was $2 + 3$.

We track dependencies between cache entries. When (symbolic execution for cache entry) A reads (the result of) B, A is registered as a consumer of B. When symbolic execution finishes and its result has changed, we place all its consumers on a global worklist. Once the worklist is empty, we have reached a fixed point. We study termination in Sec. 6.

We now discuss result initialization, then we give further details about our cache. We initialize each result in the cache to the empty relation $\bot$. Intuitively, each result then grows towards a fixed point. We interpret $\bot$ as "the result of" non-termination. We justify this interpretation as follows. The result (an input-output relation) indicates which values the call may return during ordinary execution. If the relation is empty, the call may not return any value at all. This is only possible if the call does not return.

Since our *call* node reads the cache, conceptually it can now report $\bot$, and every node needs to handle $\bot$. Continuing with the non-termination interpretation of $\bot$, most nodes report $\bot$ if any of their inputs do. Only *if* deviates from this, because *if* is non-strict. *if* reports $\bot$ if the condition is $\bot$. Otherwise we proceed as before: we attempt to decide the condition. If we succeed, we report the result of the correct branch. If we fail, morally we want to report a conditional expression (`ite` c a b), which is equivalent to:

$$y \text{ such that } (c \wedge y = a) \vee (\neg c \wedge y = b)$$

This is not a problem when we have expressions a and b from the two branches (consequent and alternate). However when a branch reports $\bot$, we *do not* have an expression for the possible values from that branch. Suppose for the sake of exposition that the alternate branch reports $\bot$. Then whenever $\neg c$, y can take *no values at all.* Symbolically:

$$\begin{aligned} & y \text{ such that } (c \wedge y = a) \vee (\neg c \wedge false) \\ =\ & y \text{ such that } (c \wedge y = a) \vee false \\ =\ & y \text{ such that } (c \wedge y = a) \end{aligned}$$

This algebraic manipulation shows that when the alternate branch reports $\bot$, we should report the result from the consequent branch, *and assert the condition to be true.* This assertion may appear strange, given the non-termination interpretation of $\bot$: how can we assert that the condition is true, "on the grounds" that the alternate branch does not terminate? However, this intuitive objection misses a subtle point. The result of the *if* node describes the possible values coming out of the *if* node, *if the if node terminates* - as all our results do. If the *if* node terminates, but the alternate branch would not terminate (i.e. it reports $\bot$), then we *do* know that the condition was true. This justifies the assertion of the condition in the result - intuitively, as well as algebraically. In the extreme, if both branches report $\bot$, *if* reports $\bot$.

The *possibility* of our cache depends on two things. First, we need *something to store in the cache.* Our call-return discipline matches a call to a future return, at which point we store a result. Second, we need a notion of equivalence of arguments, to decide what constitutes a cache hit. We use set equality, and we now describe how to check it.

The first step is to view the actual argument as a set comprehension. Suppose we have an expression e, supported by declarations ds and propositions ps. These can be viewed as the set $\{e \mid ds,\ ps\}$.

We then define set operations, building up to set equality. To simplify these definitions, we first introduce a fresh variable for e, and we quantify over ds explicitly in a new proposition. For example,

$$\{x - 1 \mid x \in \mathbb{Z},\ x > 1\}$$

becomes

$$\{y \in \mathbb{Z} \mid \exists x \in \mathbb{Z} \,.\, y = x - 1 \wedge x > 1\}$$

which in general looks like $\{x \in \mathrm{T} \mid P(x)\}$. Set complement and intersection are standard (we omit type declarations):

$$\neg\{x \mid P(x)\} = \{x \mid \neg P(x)\}$$

$$\{x \mid P(x)\} \cap \{y \mid Q(y)\} = \{x \mid P(x) \wedge Q(x)\}$$

which involves a substitution from $Q(y)$ to $Q(x)$.

We check emptiness with an SMT solver. Given $\{x \mid P(x)\}$, we declare x, assert $P(x)$, and check SATisfiability:

SAT → `inhabited`
UNSAT → `empty`
UNKNOWN → `unknown`

We define set inclusion in terms of the emptiness check:

$$S \subseteq T \iff S \cap \neg T \text{ is empty}$$

whose value can be `unknown` because of the emptiness check. We define set equality as double inclusion:

$$S = T \iff S \subseteq T \wedge T \subseteq S$$

with a ternary conjunction, since inclusion can be unknown:

$a \wedge b =$
| a or b is `false` → `false`
| a or b is `unknown` → `unknown`
| otherwise → `true`

Checking set equality like this is slow in practice, significantly for overall performance. We have another method which relies less on the SMT solver, is a thousand times faster, but depends on further assumptions. We omit it in this paper.

With polymorphic types, there is a complication with argument equivalence. Consider two inputs to a polymorphic list length program: `[x y 3]` and `[x y true]`. Ideally, we would want these to fall into the same cache entry. However, we have no clear formulation of this. We avoid the question and require equal monomorphic types for argument equivalence.

We summarize partial interpretation at a high level. To partially interpret a root call `(pgraph, args)`, we create a cache entry for it, initialize its result to $\bot$, and put it on the worklist. We then perform symbolic execution of each worklist item, to re-compute its result. Each of these re-computations may place more items on the worklist. Once the worklist is empty, we have reached a fixed point, and we read the result from the original cache entry.

We started with symbolic execution, which computes histories. We *abstracted* these histories to sets, at every call. This enabled our high-level architecture, which resembles abstract interpretation: a cache, a worklist, and a fixed point computation. We now have two tasks:

- prove that our abstraction is lossless: that partial interpretation is as accurate as symbolic execution
- study the termination of partial interpretation.

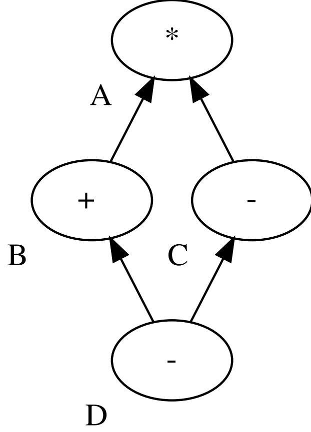

Figure 2: A diamond-shaped *pgraph* fragment.

5 IRRELEVANCE

Suppose we compute some value B, and later some value D which (transitively) depends on B. In a *pgraph*, this manifests as a path $D \rightarrow^* B$. Our main claim is that under suitable conditions, discarding the history of B (turning it into a set) loses no accuracy at D. To prove this, we must first state it precisely. It then justifies our handling of calls.

We have seen that discarding history loses accuracy in general, for example in the case of $2x + 1$ and $2x - 1$. However, we have also seen that in the familiar special case of complete information, discarding history does not lose accuracy. We now elicit another such special case.

Fig. 2 shows part of the *pgraph* for the example of $2x + 1$ and $2x - 1$. The nodes correspond to sub-expressions:

$$\text{A: } 2x \qquad \text{B: } 2x+1 \qquad \text{C: } 2x-1 \qquad \text{D: } (2x+1)-(2x-1)$$

We omit the remaining nodes: x and the constants 1 and 2.

In this example, discarding the history of B loses accuracy at D. We note these abstract features of the example:

- B and C share a past at A
- B and C share a future at D
- there is no path between B and C.

Intuitively, if B and C have no shared past, we gain nothing by preserving their histories. And their shared past is only *used* once D later rejoins B and C. We give an intuition for the third point shortly. We first state our necessary *side path condition* for the relevance of the history of B at D:

$$\exists A \text{ with } D \rightarrow^* B \rightarrow^* A \text{, and also } D \rightarrow^* A \textit{ not through } B.$$

To see the significance of the side path, suppose D only has paths to A through B. Then in any expression for D, we can *eliminate* A by rewriting in terms of B. Intuitively then, any relational information about A is captured in the relational information about B. There is a case of this in our concrete example: the difference (D) depends on x (not shown), but only through $2x$ (A). We can therefore set $u = 2x$ and reason about $(u+1)-(u-1)$, with the same final result.

This clarifies our claim's premise: if there does not exist A with $D \rightarrow^* B \rightarrow^* A$ and a side path $D \rightarrow^* A$, we claim that discarding the history of B loses no accuracy at D. It remains to clarify the conclusion: discarding history and losing accuracy. We have managed this only with the additional formalism we introduce now.

Table 1: A fragment of R_D, from Fig. 2.

...	A	B	C	D
		...		
...	0	1	-1	2
...	2	3	1	2
...	4	5	3	2
		...		

We define a *relational semantics* for the symbolic execution used in partial interpretation. Instead of mapping a *pgraph* node n to a symbolic expression, we map n to a *relation* R_n. Our relations are similar to those found in databases and formalized by Benzaken et al. [2014]. Tab. 1 shows a fragment of R_D for the $2x+1$ and $2x-1$ example from Fig. 2.

A relation is headed by a set of "columns" - its *support*. For us, this is the set of nodes reachable from n, including n. In Tab. 1, we show only the part of the support shown in Fig. 2: $\{A, B, C, D\}$. A relation contains a set of "rows". For us, a row is a possible *execution* up to n: an assignment of an ordinary value to each node in the support.

We define three relational operations. The *projection* of a row x on a set of columns S is $x|_S$, and it keeps only the columns from S in x. We lift projection to a relation R by projecting all the rows:

$$R|_S = \{x|_S \mid x \in R\}$$

Projection discards history. In particular, $R_n|_{\{n\}}$ keeps only column n in R_n, which is the set of possible values at n.

$Q \bowtie R$ is the *relational join* of relations Q and R, with supports S_Q and S_R. $Q \bowtie R$ has support $S_Q \cup S_R$ and contains the tuples which agree with both relations:

$$Q \bowtie R = \{x \mid x|_{S_Q} \in Q \land x|_{S_R} \in R\}$$

We now compute R_n by structural induction on the *pgraph*. At *const* or *prog* (or a nullary *tuple*), we report a constant: a single row with support $\{n\}$. At *entry*, we report the set of possible arguments, also with support $\{n\}$.

Everywhere else, we compute relations for our successor nodes, and also a local *template* relation L_n. L_n relates the inputs of n to its outputs. L_n has neutral column names, which we rename to target the correct *pgraph* nodes. We then join all these relations.

Everywhere except *if*, the successor relations are available directly by induction. At *if*, suppose the three inputs are c, a, b. We first compute $R_c|_{\{c\}}$, which is the set of possible conditions. We then compute R_a (and/or R_b) if `true` (and/or `false`) is possible. If a condition is impossible, in that branch we use $\bot$ instead, over the support of that branch.

Everywhere except *call*, L_n depends only on n. We show examples, where a and b are arbitrary:

- binary *tuple* (aka `mkPair`): $L_n = \{(a, b, (a, b))\}$
- *proj* 0 (aka `fst`): $L_n = \{((a, b), a)\}$
- *prim* +: $L_n = \{(a, b, a + b)\}$
- *if*: $L_n = \{(\texttt{true}, a, b, a)\} \cup \{(\texttt{false}, a, b, b)\}$.

At *call*, we get L_n with a little work. Let R_p and R_a be the relations computed for our program and argument. We project both to sets: $R_p|_{\{p\}}$ and $R_a|_{\{a\}}$. If $R_p|_{\{p\}}$ is not unique, L_n is the full relation. If $R_p|_{\{p\}}$ is unique, we look it up in the global cache, with `args` = $R_a|_{\{a\}}$. L_n is then the input-output relation read from the cache.

Our relational semantics is an *extensional* view of symbolic execution: conceptually, our relations enumerate infinitely many possible executions. In contrast, symbolic execution implements relations in *comprehension*. For example, the symbolic expression $2x + 1$ naturally corresponds to the relation $\{(x, 2x+1) \mid x \in \mathbb{Z}\}$. Comprehension is suitable for implementation, but the extensional view lets us state what we need: discarding history is projection, and losslessness will be an equality between relations.

We now set the stage for our main theorem: a *pgraph* containing a path $D \to^* B$. We execute this program up to D in two ways. First, we apply our relational semantics to get a baseline result: we compute R_D. Second, we define a modified form of our relational semantics: $M_n^{(B)}$ is computed like R_n everywhere except B. At B, we intervene: we report $R_B|_{\{B\}}$, which discards all history from B. We then continue executing.

Morally, our theorem's conclusion is that $R_D = M_D^{(B)}$. However, this is technically wrong because the two sides may have different supports. Intuitively, $M_D^{(B)}$ has been cut off from the part of the support of B to which D has no side path. This is because the only contribution of those columns to R_D came from B, but we projected them out in $M_B^{(B)}$. We note that B itself remains in the support of $M_D^{(B)}$.

We resolve this by projecting R_D on the support of $M_D^{(B)}$. This choice intuitively discards minimal information before comparing the two sides, and it passes the sanity checks we discuss next. We now state our irrelevance theorem:

Theorem 5.1. *Suppose a pgraph contains a path $D \to^* B$. If there exists no A with both $D \to^* B \to^* A$ and a side path $D \to^* A$, then $R_D|_S = M_D^{(B)}$, where S is the support of $M_D^{(B)}$.*

The equality $R_D|_S = M_D^{(B)}$ is a double inclusion. This gives us a proof method, but also two sanity checks:

(1) $M_D^{(B)}$ should never be *more accurate*, so $R_D|_S \subseteq M_D^{(B)}$ should be true regardless of any graph condition
(2) we should exhibit an example where $M_D^{(B)} \not\subseteq R_D|_S$, when we allow a side path.

We then prove $M_D^{(B)} \subseteq R_D|_S$ when there is no side path. Finally, we use Theorem 5.1 to justify our approach.

The main intuition in these proofs is that each row of our relations is a *concrete execution*. To construct one row (in R_n), at each node we select a concrete value compatible with the values selected so far. In $M_n^{(B)}$ however, we have a "free choice" at B: we can select a value compatible with *any* execution; not just the execution in progress.

We prove sanity check (1) with this intuition. Suppose we have an execution $r \in R_D$. We execute the same way in $M^{(B)}$. With our free choice at B, in particular we select the value selected in r. Therefore $r|_S \in M_D^{(B)}$.

In the other direction (2), consider again the example of $2x + 1$ and $2x - 1$ (see Fig. 2). Here $(0, 3, -1, 4) \in M_D^{(B)}$, for the support [$A$, B, C, D]. The free choice $B = 3$ is justified because there exists an execution which would produce it: $(2, 3, 1, 2)$. But $(0, 3, -1, 4) \notin R_D$ (here $R_D = R_D|_S$).

We now prove $M_D^{(B)} \subseteq R_D|_S$ when there is no side path. Suppose $m \in M_D^{(B)}$. We must exhibit $r \in R_D$ with $r|_S = m$. In m, a free choice has been made at B. But we know there exists a *pre-execution* (up to B) which would justify that free choice. We *replace* the support of B in m with that pre-execution, to get r. Because there is no side path from D to the support of B, this replacement does not invalidate the *rest* of the execution, and $r \in R_D$.

Theorem 5.1 justifies our approach as follows. Suppose B represents the argument to a call. A is any node in the caller's scope, and D is anything computed within the call. There *cannot be a side path* $D \to^* A$, because we have funnelled all the information going into a call through its argument. We have no closures, and intuitively it makes no sense for the call to read from the caller's scope, since the identity of the calling program is not even known.

Therefore, discarding the history of the arguments does not change the relation *over the entire support of* $M_D^{(B)}$: the scope of the call. Projecting further on $\{B, D\}$, the *input-output relation* is not changed. This exactly justifies our approach: indexing input-output relations by sets of arguments.

We note that Theorem 5.1 is stronger than what we use. We use only equality over the support $\{B, D\}$, but we have equality over the entire support of $M_D^{(B)}$. We do not know what to think of the gap between the two.

6 TERMINATION

We have shown that partial interpretation is as accurate as symbolic execution. We now sketch a termination argument, and we see that it fails in three places. We then address these failures in a mostly unprincipled way.

Partial interpretation creates cache entries and re-computes their results. Intuitively, if we create finitely many cache entries, and re-compute each of their results finitely many times, then we terminate in finite time. We argue each.

As it stands, infinitely many cache entries can be created. Consider `fact(x, d)`, where `d` (dead) is incremented in the recursive call but never used. Starting with `x` completely unknown and `d` = `{0}`, we create entries for `fact(ℤ, {0})`, `fact(ℤ, {1})`, etc. We discuss this problem in Sec. 6.2.

We argue in three parts that each result (an input-output relation) is re-computed finitely many times:

- *directionality*: as it is re-computed, the result becomes monotonically larger
- an endpoint in that direction: the full relation
- finitely many steps from any result to the endpoint.

To argue result directionality, we introduce a strong induction on the global (chronological) order of result re-computations. We invoke this only once, to justify that a previous re-computation respected directionality.

In a first computation, result directionality is respected because the previous result was $\perp$. In subsequent re-computations, we compare the previous and current executions of the *pgraph*, and we argue a *node directionality*: the result at each node has grown monotonically. We then observe in particular that it has grown at the exit node.

We state node directionality in terms of our relational semantics from Sec. 5:

Lemma 6.1. *(Node directionality) For some cache entry, if R_n was computed previously at node n, and R'_n is computed now, then $R_n \subseteq R'_n$.*

Proof. We proceed by structural induction. At *const* or *prog* (or a nullary *tuple*), $R_n = R'_n$. Everywhere else, we assume by induction that the relations at our successors have grown. It remains to show that $L_n \subseteq L'_n$. We state without proof that the relational join of larger relations is also larger.

Everywhere except *call*, $L_n = L'_n$. At *call*, we compute the set of arguments $A = R_a|_{\{a\}}$ and there are two cases. If $A = A'$, we read L'_n from the same cache entry E as L_n:

- if E has not been re-computed, $L_n = L'_n$
- if E has been re-computed, we invoke our strong induction hypothesis: $L_n \subseteq L'_n$.

If $A \neq A'$, we are reading from a new cache entry E', and we have no guarantee that the result there is larger than L_n. In the extreme case, E' might be freshly created with $\perp$ as a result. We discuss this problem in Sec. 6.3. □

As it stands, there can be infinitely many steps from a result to the full relation. An example is `fact(x)`, where `x` is completely unknown. We compute a first result:

$$\text{if } x \leq 1 \text{ then } 1 \text{ else } \perp$$

where we abuse notation by writing $\perp$ inside a conditional expression. Then:

$$\text{if } x \leq 1 \text{ then } 1 \text{ else}$$
$$\text{if } x = 2 \text{ then } 2 \text{ else } \perp$$

etc. We are progressively unrolling factorial into conditional expressions, and this never reaches the full relation. We discuss this problem in Sec. 6.1.

This completes our sketch of a termination argument. We now address the three places where it failed, remaining principled as far as we are able. In the next section, we take the last steps in an unprincipled way.

6.1 Limiting growth steps

The first failure is that we can compute a sequence of growing results $R_0 \subset R_1 \subset \ldots$ which never reaches the full relation. This is a feature of our lattice of possible relations: it contains *infinite ascending chains*.

We introduce an *approximation lattice*, which is included in our original lattice but has no infinite ascending chains. Whenever a result grows, we "round it up" to the nearest approximation which includes the result. This rounding is similar to the abstract interpretation notion of *widening* - we adopt this terminology.

The choice of approximation lattice determines what information is kept and discarded. This choice could conceivably be different for each call, adapting to the information needs of the caller. We speculate that this is an opportunity for further lossless abstraction, but we have not explored it. In the next section, we use a fixed approximation lattice.

6.2 Limiting cache entry creation

The second failure is that we can create cache entries indefinitely. Cache entries are created for distinct arguments to calls. Calls occur only at *execution points*: either the root call, or a *call* during symbolic execution for some cache entry. Every execution point is the endpoint of a sequence of calls beginning with the root call. Intuitively, this sequence is a path in the program's unrolled call graph. This unrolling can be infinite for a recursive program.

Additionally, several distinct arguments can occur at a single execution point. This happens if arguments grow, such as in the case $A \neq A'$ in the proof of Lemma 6.1. We limit this by widening arguments when they grow.

It remains to limit the unrolling of the program's call graph. Whenever there is a cache miss, we first make a *termination assessment*. If we judge that we may be in an infinite unrolling, we intervene with a *termination resolution*: we adjust the arguments to limit their proliferation.

We have not explored these much. We make one principled observation: if we have not called the same *pgraph* twice on the way to an execution point, there is no risk of infinite unrolling yet. We identify this by tracking a *call stack* during partial interpretation. In the next section, we supplement this with ideas from online partial evaluation [Weise et al. 1991]. In Sec. 8, we mention possibilities based on other work.

6.3 Handling congruence

The third failure is that when arguments grow at an execution point ($A \subset A'$), we read R' from a new cache entry, but we have no guarantee that R' includes the result R from the old cache entry. Intuitively, it seems that $R \subseteq R'$ *should* be true: more possible arguments should imply more possible results. We refer to this as *congruence* (of arguments and results):

$$A \subseteq A' \to R \subseteq R'.$$

Note that this generalizes our caching principle, which was:

$$A = A' \to R = R'.$$

We do not know if congruence follows from our development, or if it is a separate principle.

In practice, we are not aware of an object program where termination is broken by argument growth at an execution point. Ruf [1993, Sec. 4.4.2] describes features of programs where what we call node directionality is broken. We do not know what is additionally needed to break termination.

Nevertheless, we have explored five methods of ensuring termination when arguments grow. This exploration is complicated by the lack of an example object program. We omit one method which appears to fail, and two which are *ad hoc*. We describe two methods based on congruence.

The first method propagates *some congruence information*. When an execution point previously read R but now reads R' (from a new cache entry) with $R \not\subseteq R'$, we intervene: we first set R' to $R \cup R'$. Intuitively, we "initialize" R' with R. This intervention respects result directionality, so our overall termination argument is maintained. This method propagates exactly enough congruence information to ensure node directionality.

The second method takes this further and propagates *all congruence information*. When we create a new cache entry E, we initialize its result to the union over all cache entries E' with $E' \subseteq E$. When we re-compute a result for E, we union it into all cache entries E' with $E \subseteq E'$. The intuition is that congruence is not just an obstacle in the termination argument; it is an *opportunity* to reach a fixed point faster.

These methods imply that results no longer change only by re-computation; they also change by virtue of congruence. We do not know how this affects *what we are computing*. In particular, we do not know if the same result is computed by propagating some or all congruence information.

7 PARTIAL EVALUATION

We apply our approach to partial evaluation (automatic program specialization), which was the original motivation for this work. We explain this motivation.

In the best conditions, partial evaluation has been shown to specialize programs with great effect [Andersen 1996; Berlin 1989; Mossin 1993]. However, partial evaluation folklore says that these best conditions require manual adjustment of input programs, because a given partial evaluator is not uniformly accurate over all input programs. We ask what would happen if a partial evaluator *was* uniformly accurate, and readily available to programmers (e.g. as part of a compiler).

Routinely, ordinary programmers specialize general libraries by hand, for specific use. This specialization could be automated by partial evaluation, saving costs and errors. Flipping this around, we could write more general programs in the first place: we would not be so concerned about a later penalty in performance, or the cost of specialization. Partial evaluation could be a mainstream programming tool.

We therefore propose a principled approach to partial evaluation, aimed at complete accuracy. In this sense, we continue past work on *online* partial evaluation [Berlin 1989; Katz and Weise 1992; Katz 1998; Ruf 1993; Weise et al. 1991]. Online methods use all available information, and are more accurate than *offline* methods.[2] We take this further by incorporating modern symbolic execution, abstract interpretation, and automatic deduction (e.g. SMT solvers).

Program specialization has a strikingly simple formulation in terms of partial interpretation. The classic dichotomy of `known` vs `unknown` inputs is a special case of a set of possible inputs `args`. Then, the main question during specialization is whether we know the result of an operation, so we can strip out that operation. To decide this, first perform partial interpretation for `(pgraph, args)`, reaching a fixed point. We then have R_n for every node n within the call. Now check if $R_n|_{\{n\}}$ - the set of possible values for n - is a singleton.

[2] Ruf [1993, Chap. 2] argues this convincingly.

We specialize a *pgraph* by "copying" it, starting at the exit node, with three special rules. If a result is unique, we write a *const* (or *prog*). At *if*, if the condition is unique, we skip the *if* and write only one branch. At *call*, if the called program is unique, we write a *call* to the recursively specialized program. Any termination difficulties have already been resolved during partial interpretation.

This formulation places partial evaluation in a broader context: if we can compute accurately over partial information (in theory), we have the best chance of delivering uniformly accurate partial evaluation (in practice). Separating specialization from partial interpretation also improves specialization accuracy, as we illustrate in Sec. 7.3.

In Sec. 6, we saw that partial interpretation needs additional pieces to ensure termination:

- widening of results (Sec. 6.1)
- widening of arguments (Sec. 6.2)
- termination assessment and resolution (Sec. 6.2)
- handling of congruence (Sec. 6.3).

Since we do not have principled solutions, we fill in these blanks as simply as we can for partial evaluation. We reduce the widening of results to the widening of arguments, by first projecting a result to a set. As an exception, we do not widen a result at all when it is first computed (i.e. the previous result was $\perp$). This preserves the input-output relation for many results in practice. We ignore the problems with congruence, since they have not arisen in practice. We adapt an approximation lattice from Katz and Weise [1992] for our widening, and we re-use the termination assessment and resolution from Weise et al. [1991].

7.1 Widening

Our approximation lattice for widening intuitively:

- preserves uniquely known "scalars" (booleans, integers, and programs)
- discards all information about non-unique scalars
- preserves the known shape of data structures.

Another intuition is that a widened set is a value expression possibly containing the special *any* token anywhere. For example, the set of lists headed by `1` is written:

`cons(1, any)`

The set of lists of length 2 is written:

`cons(any, cons(any, nil))`

An ordinary value is a special case without *any*, which reflects the embedding of complete information as a singleton set. The pair `(any, any)` is written *any*. We omit the details of computing the right widened set.

This approximation lattice has no infinite ascending chains. Intuitively, when we grow from one widened set to another, either we turn one of finitely many unique scalars into *any*, or we shorten a list of finite length.

7.2 Termination assessment/resolution

We start from the idea in Sec. 6.2: there is only cause to adjust arguments if we have called the same *pgraph* twice on the way to an execution point. We refine this condition. We adjust arguments

only if there has been an *undecided branch* between the current call and the last call to the same *pgraph*.

This criterion is known as *guarded recursion*: the recursion is guarded by the undecided branch. Guarded recursion allows unrolling loops controlled by known values, which is critical for partial evaluation. However, it only ensures termination *when the program would terminate at run-time.* To identify guarded recursion, we place an IF marker on the call stack every time we pass an undecided branch.

In the case of guarded recursion, we ensure that sufficiently long paths (through the unrolled call graph) produce a cache hit. Observe that any infinite path contains a cycle through some point p. Consider the sequence of arguments each time we go through p. We ensure that this sequence is growing directionally towards the full set, by taking the union of the current arguments and those from the most recent call on the stack. We then widen the arguments, which ensures this reaches the full set in finitely many steps.

7.3 Uniqueness at a distance

We show a small synthetic example which illustrates the specialization accuracy we have gained.

Previous partial evaluators focused on the uniqueness of values, *locally*: if all the inputs to an operation are unique, then its result is unique - but this was the only time the result was considered unique. This is sufficient accuracy in some cases, but not all.

Instead, our approach propagates the partial information about values, "far" across the program. This enables us to find that a result is unique, *remotely.* For example, it is a mathematical identity that:

$$(x+1)^2 = x^2 + 2x + 1$$

Consider the program:

$$p(x, y) = (x+1)^2 - x^2 - xy$$

Suppose $y = 2$ and x is completely unknown. $p(x, 2) = 1$, even though none of its immediate sub-expressions have a unique value. Our approach finds this specialization.

7.4 First Futamura projection

We report on an experiment to validate our approach, then we comment on performance: the execution time of partial interpretation and specialization.

We perform the *first Futamura projection* [Futamura 1971]: we specialize an interpreter for a small language Q of arithmetic calculations. Our Q language is a restriction of *pgraph*s, expressed as *pgraph* data. Our Q-interpreter spans some 200 lines in a surface language for *pgraphs*, including many comments. We write our Q-interpreter in good faith, without trying to manipulate the results of the specializer: we use polymorphic `map`, we memoize the result at each node, etc.

We achieve a perfect first Futamura projection: given a known Q-program, all the internal logic and data structures of the interpreter disappear, and we effectively translate from Q to a *pgraph.* This reproduces a classic partial evaluation result, without manual adjustment of the input program.

7.5 Performance

Our initial implementation performed poorly: we could not finish specializing our Q-interpreter for a 1-op Q-program. We have improved performance enough to specialize for 120 ops in about two minutes. We cannot fit our entire study of performance in this paper. We mention some important points.

Execution time is dominated by waiting for answers from the SMT solver. Our most significant performance improvements reduce the SMT workload, ideally avoiding SMT queries completely.

SMT queries occur mainly during cache lookup at a *call*, where we check if sets of arguments are equal. Initially, one set of symbolic arguments could occupy several megabytes, which overwhelms the SMT solver: it fills 8 gigabytes of RAM then pages to disk. Although a set conceptually has no history, set comprehension contains a select history (an expression) which *could* have produced the set's elements. This history grows large if it extends to the start of the program.

We address this by widening arguments at every call, even when not required for termination. This sacrifices cross-call accuracy in principle, but results are not affected for these examples. Argument size is reduced, and we specialize for 1 op in 15s; for 2 ops, in 45s.

Once all arguments are widened, we can compare them syntactically to avoid SMT queries. The bottleneck then becomes widening itself, which queries the SMT solver to identify the known heads of expressions.

We address this by adopting a discipline which keeps known heads readily available in our expressions. Intuitively, we never write "y such that $y = 1$"; we write 1 instead. This discipline lets us avoid SMT queries for many widening steps, at no loss in accuracy. Maintaining this discipline requires several changes to our algorithms, which we omit here.

We achieve a roughly thousandfold speedup, at which point we specialize for 120 ops in about two minutes. Peak memory usage is about 1.2 gigabyte. Execution time is empirically quadratic in the number of operations in the Q-program.

Our implementation is written in Gambit Scheme [Feeley 2019], which compiles to `C`. We measure on an Intel i5-6600[3] running Linux 5.12. We use the SMT solver CVC4 [Barrett et al. 2011], set to timeout with UNKNOWN after 20ms for each query.

80% of execution time is spent in partial interpretation, and 20% in the subsequent specialization phase. Half of the overall time is spent waiting for answers from SMT. A quarter is spent in the Gambit Scheme garbage collector. The remaining quarter is spent in our own algorithms: mostly widening, and some syntactic comparisons.

We believe performance can be improved further, by avoiding more SMT queries and by re-implementing in a language with manual memory management (such as `C` or `C++`).

8 RELATED WORK

8.1 Online partial evaluation

Our work is influenced by the early 1990s work on online partial evaluation at MIT and Stanford, starting with Berlin [1989]. Berlin performs partial evaluation of a functional subset of Scheme, but does not handle recursion. He applies his system to numerical

[3] 3.3GHz quad core. We use only one core.

programs, which he reports as very successful, especially targeting supercomputers. His system performs a kind of symbolic execution which constructs and accesses into symbolic data structures of known shape.

Weise et al. [1991] perform automatic online partial evaluation of recursive programs, for the first time. They introduce the termination assessment and resolution we use in Sec. 7.2, and the idea of tolerating non-termination of the partial evaluator, when the object program might not terminate at run-time.

Ruf [1993] introduces the general architecture we use: a cache, a worklist, and a fixed point computation. He does this to compute more accurate results from sub-calls, rather than consider them completely unknown as previous partial evaluators did. We consider this a first trace of what we call partial interpretation. However, Ruf does not separate partial interpretation and specialization, algorithmically or conceptually. In a single phase, a fixed point is found over both results and specializations.

Ruf's work is complicated by the perceived necessity of handling closures, which reflects the lineage of his work back to Berlin: the partial evaluation of Scheme. Instead, we allow programming language design to proceed cooperatively with the design of partial interpretation and specialization.

Ruf provides some pieces of our termination argument, and in particular he identifies the problems with what we call congruence in Sec. 6.3. He suggests a solution which appears logically equivalent to our suggested approach of propagating all congruence information. However, Ruf's solution does not leverage congruence to reach a fixed point faster.

Ruf also studies a notion of argument equivalence broader than exact equality, based on tracking the information used by specialization. This allows more calls to fall into the same cache entry, which improves the performance of his algorithms. It should be possible to adapt this to our work.

Katz [1998] suggests many ideas, but we are not sure what their proper place is. For us, his main idea is tracking the information used not just by specialization as Ruf does, but also by what we call partial interpretation. Katz suggests that we can determine what information is relevant at every call and return, which is exactly what we would need for adaptive widening (see Sec. 6.1).

Katz then suggests (this is arguably his main thesis) that we can assess termination based on information use. We suspect this is backwards: intuitively, it seems we only know what information is relevant *after* we have taken a stand on termination, and possibly intervened to ensure it.

8.2 Other work

We do not have the expertise to give a deep comparison to symbolic execution and abstract interpretation. Our idea of adaptive widening (see Sec. 6.1) appears related to the notion of *a posteriori soundness* [Might and Manolios 2009] in abstract interpretation. Separately from this, Nguyen [2019] uses an SMT solver for symbolic execution. Most significantly for us, Nguyen uses *size-change graphs* [Ben-Amram and Lee 2007] to assess termination of execution paths. This appears to be a more principled approach to termination assessment than guarded recursion.

We view offline partial evaluation as complementary to our work. We focus on accuracy, whereas offline methods [Jones et al. 1993] pre-compute part of their work to achieve faster specialization and self-specialization. In principle, the two approaches could be reconciled: pre-compute all you can, without sacrificing accuracy.

We understand our program representation as a culmination of static single assignment (SSA [Rosen et al. 1988]) representations. Starting from a control flow graph (CFG) which totally orders execution, the SSA transformation manifests data dependencies ("use-def" connections). These data dependencies completely characterize many values, but not those which depend on a branch in the original CFG. Part of the original control flow is therefore preserved in SSA, and the Φ node selects between values based on which control path is taken.

Instead, our *if* node is connected to an explicit boolean condition. This is simpler to analyze, which is our main goal. Additionally, it makes the remaining control flow in SSA unnecessary, so we discard it. In a *pgraph*, execution is partially and minimally ordered by data dependencies.

In this sense, *pgraph*s are similar to value dependence graphs [Weise et al. 1994] - but we have no closures and we inject different primitives; and to program expression graphs [Tate et al. 2009] - but we have no imperative looping constructs.

We have focused on SMT solvers as examples of modern automatic deduction. Automated theorem proving [Kovács and Voronkov 2013] is an alternative. By extension, we also mention modern program optimization [Tate et al. 2009] and computer algebra, which manipulate symbolic expressions towards certain goals.

9 CONCLUSION

We have taken a step towards principled computation over partial information. We have distinguished several notions of partial information. We have shown that (for our object programs) history can be discarded from the arguments to a call, at no loss in accuracy. This is not enough to ensure termination, but it brings us closer to the principled accuracy and termination we seek.

We have separated partial evaluation into partial interpretation and specialization. We have reproduced a classic result from partial evaluation, without manual adjustment of the input program. On a synthetic example, accuracy is improved over previous partial evaluators. Performance is a challenge, but we have improved it a thousandfold, and we believe more is possible.

Our experiments are rudimentary, but we achieve good results *despite* the many limitations in our work. We inventory these limitations in the next subsection, and they suggest the following plan. Addressing limitations on object programs enables more realistic experiments. Every time one experiment succeeds, we are one step closer to bringing these methods to the mainstream - in particular, uniformly accurate partial evaluation. Every time one experiment fails, we can *diagnose* the failure in terms of known limitations, which directs our principled work.

9.1 Limitations and future work

Our object programs lack algebraic data types and some primitive types: machine words, and strings. We anticipate no difficulty with these additions. Our object programs also lack external effects,

mutation, and non-local exits (e.g. continuations). These require fundamental changes to *pgraph*s, and the challenge is to integrate them with partial interpretation.

The study of congruence (see Sec. 6.3) is not finished. We should exhibit a concrete example of non-termination due to argument growth. We should also compare the results of different ways of ensuring termination in this context.

We have limited the accuracy of partial interpretation in several ways. First, we widen arguments at every call. This is theoretically necessary in some places, for termination; and practically desirable everywhere, for performance. Our widening is unprincipled. Principled, adaptive widening should determine what information is relevant or not, at each call and return. This may build on [Katz 1998].

Second, our termination assessment (guarded recursion - see Sec. 6.2) is unprincipled. A more principled alternative should be possible with size-change graphs [Ben-Amram and Lee 2007], as in [Nguyen 2019]. We also mention the linear algebra techniques of [Tiwari 2004].

Third, our language of propositions avoids recursion, to match what we perceive to be the requirement of current automatic deduction, e.g. an SMT solver. We do not have the expertise to judge the flexibility of this requirement.

Fourth, we do not track path conditions; and fifth, we do not analyze calls when the called program is not uniquely known (see Sec. 3). We anticipate no serious difficulty with either.

We have limited the performance of partial interpretation in several ways, but we omit most of that discussion in this paper. We re-iterate the need to reduce reliance on the SMT solver, and the idea of broader equivalence of arguments (see Sec. 4 and [Ruf 1993]). Broader argument equivalence would reduce the number of cache entries, which is central to performance.

Finding unique results does not account for all of program specialization. Our methods should be supplemented by principled program optimization, as in e.g. [Tate et al. 2009].

We have applied partial interpretation to partial evaluation. We end with another possible application: automatic program verification. Program verification deals in assertions, pre-conditions, post-conditions, and invariants. All these can be seen as sets/relations within which actual data must stay confined. Partial interpretation computes sets/relations which encompass actual data. If these are included in the asserted sets/relations, then those assertions have been verified. This places program verification in the same broader context as partial evaluation: computation over partial information.

ACKNOWLEDGMENTS

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada grant N° 298311/2012 and RGPIN-2018-06225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

REFERENCES

Peter Holst Andersen. 1996. Partial evaluation applied to ray tracing. In *Software Engineering im Scientific Computing*. Springer, 78–85.

Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*. Springer, 171–177.

Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, Vol. 13. 14.

Amir M Ben-Amram and Chin Soon Lee. 2007. Program termination analysis in polynomial time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 1 (2007), 1–37.

Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. 2014. A Coq formalization of the relational data model. In *European Symposium on Programming Languages and Systems*. Springer, 189–208.

Andrew A Berlin. 1989. A compilation strategy for numerical programs based on partial evaluation. (1989).

Marc Feeley. 2019. Gambit v4.9.3 manual. http://www.iro.umontreal.ca/~gambit/doc/gambit.pdf.

Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, computers, controls* 2, 5 (1971), 45–50.

Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Peter Sestoft.

Morry Katz and Daniel Weise. 1992. Towards a New Perspective on Partial Evaluation. *PEPM* 92 (1992), 19–20.

Morris Joel Katz. 1998. *A new perspective on partial evaluation and use analysis.* Ph.D. Dissertation. Stanford.

Laura Kovács and Andrei Voronkov. 2013. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*. Springer, 1–35.

Matthew Might and Panagiotis Manolios. 2009. A posteriori soundness for non-deterministic abstract interpretations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 260–274.

Christian Mossin. 1993. Partial evaluation of general parsers. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 13–21.

Phúc Nguyen. 2019. *Higher-order Symbolic Execution.* Ph.D. Dissertation.

Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 12–27.

Erik Steven Ruf. 1993. *Topics in online partial evaluation.* Ph.D. Dissertation. Stanford.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 264–276.

Ashish Tiwari. 2004. Termination of linear programs. In *International Conference on Computer Aided Verification*. Springer, 70–82.

Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. 1991. Automatic online partial evaluation. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 165–191.

Daniel Weise, Roger F Crew, Michael Ernst, and Bjarne Steensgaard. 1994. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 297–310.