Université de Montréal

Computation over partial information

A principled approach to accurate partial evaluation

par

Ian Sabourin

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

July 20, 2021

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Computation over partial information

A principled approach to accurate partial evaluation

présenté par

Ian Sabourin

a été évalué par un jury composé des personnes suivantes :

Michel Boyer

(président-rapporteur)

Marc Feeley

(directeur de recherche)

Stefan Monnier

(codirecteur)

Miklós Csűrös

(membre du jury)

Résumé

On est habitué à penser comme suit à un programme qui exécute: une donnée entre (un *input*), un moment passe, et un résultat ressort. On assume tacitement de l'information complète sur le *input*, le résultat, et n'importe quels résultats intermédiaires.

Dans ce travail-ci, on demande ce que ça voudrait dire d'exécuter un programme sur de l'*information partielle*. Comme réponse possible, on introduit *l'interprétation partielle*, notre contribution principale. Au lieu de considérer un seul *input*, on considère un ensemble de *inputs* possibles. Au lieu de calculer un seul résultat, on calcule un ensemble de résultats possibles, et des ensembles de résultats intermédiaires possibles.

On approche l'interprétation partielle à partir du problème de la *spécialisation de programme*: l'optimisation d'un programme pour certains *inputs*. Faire ça automatiquement porte historiquement le nom d'*évaluation partielle*. Ç'a été appliqué avec succès à plusieurs problèmes spécifiques. On croit que ça devrait être un outil de programmation commun, pour spécialiser des librairies générales pour usage spécifique - mais ce n'est pas le cas.

Souvent, une implantation donnée de l'évaluation partielle ne fonctionne pas uniformément bien sur tous les programmes. Ça se prête mal à un usage commun. On voit ce manque de régularité comme un problème de précision: si l'évaluateur partiel était très précis, il trouverait la bonne spécialisation, indépendamment de notre style de programme.

On propose donc une approche de principe à l'évaluation partielle, visant la précision complète, retirée d'exemples particuliers. On reformule l'évaluation partielle pour la baser sur *l'interprétation partielle*: le calcul sur de l'information partielle. Si on peut déterminer ce qu'on sait sur chaque donnée dans le programme, on peut décider quelles opérations peuvent être éliminées pour spécialiser le programme: les opérations dont le résultat est unique.

On définit une représentation d'ensembles qui ressemble à la définition en compréhension, en mathématiques. On modifie un interpréteur pour des programmes fonctionnels, pour qu'il calcule sur ces ensembles. On utilise un *solver* SMT pour réaliser les opérations sur les ensembles. Pour assurer la terminaison de l'interpréteur modifié, on applique des idées de l'interprétation abstraite: le calcul de point fixe, et le *widening*. Notre implantation initiale produit de bons résultats, mais elle est lente pour de plus gros exemples. On montre comment l'accélérer mille fois, en dépendant moins de SMT.

Mots-clés: Interprétation partielle, Évaluation partielle, Exécution symbolique, Interprétation abstraite

Abstract

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.

In this work, we ask what it would mean to execute a program over *partial information*. As a possible answer, we introduce *partial interpretation*, our main contribution. Instead of considering a unique input, we consider a set of possible inputs. Instead of computing a unique result, we compute a set of possible results, and sets of possible intermediate results.

We approach partial interpretation from the problem of *program specialization*: the optimization of a program's execution time for certain inputs. Doing this automatically is historically known as *partial evaluation*. Partial evaluation has been applied successfully to many specific problems. We believe it should be a mainstream programming tool, to specialize general libraries for specific use - but such a tool has not been delivered.

One common problem is that a given implementation of partial evaluation is inconsistent: it does not work uniformly well on all input programs. This inconsistency makes it unsuited for mainstream use. We view this inconsistency as an accuracy problem: if the partial evaluator was very accurate, it would find the correct specialization, no matter how we present the input program.

We therefore propose a principled approach to partial evaluation, aimed at complete accuracy, removed from any particular example program. We reformulate partial evaluation to root it in *partial interpretation*: computation over partial information. If we can determine what we know about every piece of data in the program, we can decide which operations can be removed to specialize the program: those operations whose result is uniquely known.

We represent sets with a kind of mathematical set comprehension. We modify an interpreter for functional programs, to compute over these sets. We use an SMT solver (Satisfiability Modulo Theories) to perform set operations. To ensure termination of the modified interpreter, we apply ideas from abstract interpretation: fixed point computation, and widening. Our initial implementation produces good results, but it is slow for larger examples. We show how to speed it up a thousandfold, by relying less on SMT.

Keywords: Partial interpretation, Partial evaluation, Symbolic execution, Abstract interpretation

Contents

List of tables

List of figures

List of abbreviations

ADT	Algebraic data type
CFG	Control flow graph
IR	Intermediate representation
JIT	Just-in-time
PEG	Program expression graph
SMT	Satisfiability modulo theories
SSA	Static single assignment
VDG	Value dependence graph

Acknowledgements

I thank my research advisors Marc Feeley and Stefan Monnier, for their time and considerable experience, and for assigning funding to me.

I thank Stefan especially, for being there the whole time, through all the ideas that became this work. I thank Marc for challenging me to question and improve my earlier research ideas.

I enjoyed my time with many fellow students. Jean-Alexandre Barszcz and Simon Génier always seemed to be around the most.

Preface to the edited version

I have made changes throughout this text, to address the comments of the evaluation committee. This preface gives insight into how this work took place, to hopefully address the main criticisms: a lack of formality overall, and a lack of direction in some parts. I agree that more formality is good, and wherever it is lacking, that is because I didn't have it.

This work started as an exploration: I wanted to see how good partial evaluation can be, and I soon had the idea that it comes down to what we know about every piece of data. I wanted to root this reasoning in modern automatic deduction. SMT seemed like a good candidate since it was getting results in many fields. I wanted to "start from scratch", to not get trapped in the premises of previous work. These premises are often not explained, and sometimes not even mentioned. By building everything myself, I would *understand* the premises, i.e. I would have some constraints on what a solution "must" be.

One of the first questions was: what does this mean, "what we know about data"? In other words, what is partial information? I tried many formulations of this, throwing one out every couple of weeks. Eventually I realized that one formulation was a set of possible values. It became a de facto working assumption that partial information is a set. Now, although I was seeking a principled formulation, I also wanted to implement this. So this set notion was not just something on paper; it had to be a data structure in a real program, and SMT had to understand it. I came up with a syntactic representation of sets in comprehension. Extended comprehension reduced the size of the data structures, compared to introducing an explicitly quantified variable for every intermediate result. Basic comprehension simplified set complement and variable naming across calls. This seemed like a good principled start.

I knew more was needed for a full solution: dealing with termination. The purpose of the termination argument is to show us where we need to intervene to get termination. I did not (and still do not) have principled solutions for these interventions. My interest was twofold: first, to outline the design space for these interventions, so that future work (mine or otherwise) can seek principled solutions *within* that design space; second, to come up with simple *un*principled solutions (many adapted from past work), so that I could have a working implementation, i.e. one point in the design space.

The implementation eventually worked, after a big false start: SMT gave poor results for dynamically typed object programs. I started over with static types. This piece of experience is reduced to a one-liner in the main text, for brevity, even though it cost me over a month of work. Nevertheless, throughout the text I hope to convey (some of) what failed as well as what worked, so that whoever reads this *does* understand the reasons for my choices/premises, and does not need to repeat such failures to find out.

Anyway, I had achieved my basic goal: integrating partial evaluation and modern automatic deduction. To my knowledge, this is the first work to do so. It was time to write this up. Very late in this writing (late June 2021), I and Stefan started realizing that sets were *not* all we were using. There was another notion in play, which somehow carried more information than sets, and captured relational information (entanglement?) between multiple data. At first it was difficult to even point at this notion, because of a pun in the implementation: the same data structure for comprehensions expressed both sets and ???. The implementation was subtly converting from ??? to a set in the conversion from extended to basic comprehension. I proposed to think of ??? as a computational history. Surprisingly, the conversion seemed intuitively fine to us, even though viewing a history as a set loses accuracy in general. I started looking into why this would be justified here, and I came up with the side path condition on *pgraph*s. The fact that it is justified was left as an informal conjecture in this text. I scrambled as I could to adjust the rest of the exposition, to account for the two notions of partial information.

After initially turning in this memoir, I worked on an article that I presented at IFL 2021. I felt a conjecture was not good enough for that venue. To prove the conjecture, I first had to state it precisely, and I did not even have a suitable language to do so. I came up with a relational semantics for partial interpretation, definitions of discarding history and losslessness, and a proof - the conjecture became a theorem. The relational semantics also cleaned up the termination argument. I refer the interested reader to the article [**SMF21**].

This work started as an exploration, with a goal of a principled understanding and a working implementation, and eventual formality. Some formality was achieved in the end, and reported in the article cited above. But in my view, formality cannot be the starting point of an exploration; formality must come later, so that it makes precise *the right things.*

If I were to rewrite this memoir, I might choose a different presentation, more in line with the article. But then, the memoir would lose the genuine quality of reflecting the work as it happened: a journey from exploration to formality.

Ian Sabourin (September 2021)

Chapter 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.

In this work, we ask what it would mean to execute a program over *partial information*. As a possible answer, we introduce *partial interpretation*, our main contribution. Instead of considering a unique input, we consider a set of possible inputs. Instead of computing a unique result, we compute a set of possible results, and sets of possible intermediate results.

We approach partial interpretation from the problem historically known as *partial evaluation*: automatic program specialization. We introduce program specialization next. Then, we progressively introduce our approach to program specialization, with partial interpretation as its main sub-problem: computation over partial information.

1.1. Program specialization

We start from a simple example. Suppose we want to compute x^n, where n is a positive integer. A standard, efficient algorithm for this is so-called fast exponentiation, where we recursively examine the parity of n:

- if $n = 0$, $x^n = 1$
- if n is odd, $x^n = x * x^{n-1}$
- if n is even, $x^n = (x^{n/2})^2$.

For general x and n, this is as good as it gets: we compute x^n with between $log_2(n)$ and $2 * log_2(n)$ multiplications, and as many parity checks.

Now suppose that in a particular application, we know we will only use this for $n = 8$. Then we would not write the algorithm as above. We can do better: square x three times, without making any parity checks. In some sense, this is still an exponentiation algorithm. It's just (much) less general than before. It is now *wrong* for any n other than 8. In return, it is more optimized for $n = 8$. In other words, it is *specialized* for $n = 8$.

Partial evaluation is the historical name for *automatic* specialization. Given the general exponentiation algorithm, and the information that $n = 8$, can we mechanically produce the specialized (optimized) algorithm which squares x three times? And of course, can we do this in general: for any program, and any information about its inputs.

In the case of exponentiation, this is definitely possible - it is a standard first example of partial evaluation. In general, the answer is less clear. We examine the track record of partial evaluation, and it informs our approach.

1.2. The appeal of partial evaluation

Partial evaluation has been applied successfully to many specific problems. For example, in ray tracing [**And96**], we can suppose the contents of a scene are known, but the position of the view point is unknown. Once the ray tracer is specialized for the known scene, it can render that scene more efficiently from any number of view points.

In parsing [**Mos93**], we start from a general parser which takes two inputs: a grammar for a language, and a string to be parsed. We then suppose the grammar is known, but the string is unknown. Once the general parser is specialized for the known grammar, it can parse any number of strings more efficiently.

This pattern of known vs unknown data is surprisingly common, and more applications are mentioned in e.g. [**JGS93**] (their section 1.2.3).

These specific successes are great, but we think the real win should be elsewhere. Routinely, ordinary programmers find that a general library is not sufficiently optimized for their specific use case. They then face a decision: accept the performance penalty; or specialize the library *by hand*, which takes a lot of time and effort. And this tension between generality and performance is usually accepted as natural: we evaluate the need for performance, and if the need is sufficient, we hand-code a specialized version.

This tension could conceivably be removed by partial evaluation. Consider the implications of having partial evaluation readily available, as part of a compiler. Libraries would get specialized automatically, saving programmers time and effort, not to mention errors. Flipping this around, we could write more general programs in the first place, because we would not be so concerned about losing performance to generality. Partial evaluation should be a mainstream programming tool.

1.3. A difficulty with partial evaluation

Unfortunately, the specific successes of partial evaluation have not carried over to a mainstream programming tool. There is no standard explanation for this, despite partial evaluation dating back to the 1970s, with a peak in the 1990s - see [**Ruf93**] and [**JGS93**] for a balanced view of some important developments.

One common problem is that a given implementation of partial evaluation does not work uniformly well on all input programs. Traditionally, this problem is transferred to the user. The user is expected to learn which kinds of programs are handled well by the partial evaluator. Then, the user is encouraged to manually adjust the input program, to (sometimes radically) improve the results of partial evaluation. This is usually phrased as "improving" the input program, but we see it as completely breaking the promise of partial evaluation: *automatic* program specialization.

This has been tolerated for specific uses of partial evaluation, in what has been essentially a laboratory setting. Indeed, the last section cited two cases (ray tracing and parsing), and in each case the researchers manually adjusted the input program to improve the results.

For general use as a programming tool, we need to handle all kinds of libraries and programs, written by all kinds of programmers, in all kinds of settings. They cannot be expected to conform to the additional needs of a partial evaluator. Partial evaluation is a good thing if it *removes* the tension between generality and performance; not if it adds a *new* tension, between the program you wanted to write, and the program the partial evaluator wants. Programming is difficult enough as it is.

Instead, we believe the inconsistency problem of partial evaluation should be addressed directly. We argue that this inconsistency is an accuracy problem. Conceptually, if the partial evaluator was very accurate, it would find the correct specialization, no matter how we present the input program.

Work has been done on improving the accuracy of partial evaluation, but usually reactively, targeting certain classes of example programs. This has not delivered the general, uniform accuracy needed for mainstream programming.

1.4. Our approach to partial evaluation

We propose a principled approach, aimed at complete accuracy, removed from any particular example program. As such, our strategy is not to build incrementally on past work. Nevertheless, we re-use useful ideas, and we highlight them in Chapter 5.

We begin by re-examining the problem statement. Informally, there are two inputs to partial evaluation: a program, and information about its inputs. We need to clarify what this means. Chapter 2 discusses our representation of programs. For now, we discuss the notion of information about the inputs.

In the traditional presentation of partial evaluation, we consider a program of several inputs - most simply, of two inputs x and y. We then suppose that the value of x is known, whereas the value of y is unknown. The question is then: how can the program be specialized (optimized), knowing the value of x?

We generalize this dichotomy of known vs unknown inputs and allow arbitrary specification of the inputs. For example, we may wish to specify that $x > 0$, or that $x < y$. Most generally, we allow specifying the *set* of possible inputs (x, y). Chapter 3 discusses how to express such a set. The basic question remains the same: how can the program be specialized, knowing the input specification?

Conceptually, the bulk of specialization comes down to one question: which operations can we remove? Removing an operation is good, because that makes the program more efficient. And it's *possible* to remove an operation, if we know its result.

To decide which results we know, we forget (for now) the *goal* of specializing a program, and focus on the set of possible inputs. Such a set constitutes *partial information* about the inputs. During ordinary execution, we have complete information: an input has a single value taken from a data domain, for example the integers $\mathbb{Z}$. In contrast, partial information about an input is a set of possible values, over the same domain.

We can extend this notion, to partially known results from every operation in the program. That is, we can consider a set of possible values, for each result from every operation. In doing this, we are effectively introducing a new domain: the power set of the original domain. During ordinary execution, each (completely known) value is taken from (e.g.) $\mathbb{Z}$; in this new setting, each (partially known) value is taken from the power set of $\mathbb{Z}$ - each value is a subset of $\mathbb{Z}$. We refer to this domain of sets as the *partial information domain.* Complete information becomes a special case in this partial information domain: a singleton set indicates a single possible value. In other words, the original domain is naturally embedded in the partial information domain.

We can then ask what it would mean to execute the program over this partial information domain. And here we are, at *partial interpretation*: computation over partial information. Supposing we can do this, we can return to our original goal of program specialization. We can decide which operations can be removed from the specialized program: those whose result is uniquely known - a singleton set.

Our approach to partial interpretation is detailed in the next several chapters. If we can compute accurately over partial information (in theory), we have the best chance of delivering generally applicable partial evaluation (in practice). Our theoretical and practical goals are therefore aligned, and this is the driving force throughout this text.

Before we move on to this, we give an informal preview of the most accessible parts of our approach. We will represent a set of possible values as a set comprehension, from mathematics. This looks like $\{x \mid P(x)\}$: the set of objects x that satisfy a predicate P. We will compute one of these sets for each piece of data in a program.

One simple and illustrative case is arithmetic addition. Suppose we have two sets of integers: $\{x \mid P(x)\}$ and $\{y \mid Q(y)\}$. Then their sum is $\{x + y \mid P(x) \wedge Q(y)\}$.

In other cases, computing over these sets will be more like logical reasoning. We will check if a set contains a unique member, or if a set is equal to another set, etc. It's easy to understand one reason to check that a set contains a unique member: these are the sets whose producing operations we want to remove from the program, to specialize it.

To reason about sets, we will use modern automatic deduction, specifically an SMT solver (Satisfiability Modulo Theories - [**DMB08**], [**BCD$^+$11**]). SMT solvers are the result of a long line of work going back to the boolean SAT(isfiability) problem. They have recently enjoyed success in many practical applications. We plan to benefit from this state of the art.

We show a brief example, to give a sense of what SMT can do. Suppose we have a set of pairs of integers (x, y), with $x^2 - 2x + 1 < y < -x^2 + 2x + 1$. Is this inhabited? Does it have a unique member? We issue the following commands to the SMT solver:

```
(declare-const x Int)
(declare-const y Int)
(assert (and (< (+ (* x x) (* -2 x) 1) y) (< y (+ (* (- x) x) (* 2 x) 1))))
(check-sat)
```

The SMT solver reports that the constraints are SAT(isfiable), so the set is inhabited.

```
(get-model)
```

The SMT solver exhibits a point in the set: $(1, 1)$.

```
(assert (or (distinct x 1) (distinct y 1))
(check-sat)
```

The SMT solver reports UNSAT, and we know the original set has the unique member $(1, 1)$. These are routine queries for an SMT solver. We will use all this and more, to compute over partial information. Then, we will use the results to perform automatic specialization.

1.5. Our contribution

To our knowledge, this work is the first to integrate partial evaluation with modern automatic deduction. We make the following specific contributions:

- we clearly separate partial evaluation into specialization and partial interpretation (computation over partial information)
- we walk through how to do both, in theory and in practice
- we give an informal termination argument for partial interpretation
- we formulate partial information to ground it in modern automatic deduction (specifically, an SMT solver)
- we show how to reduce reliance on the SMT solver, to improve performance by three orders of magnitude.

1.6. Organization of this document

First, we need a system of programs over which to develop partial interpretation. Chapter 2 describes a graph representation of programs. We discuss our design decisions for this representation, and we describe an ordinary interpreter for it.

Chapter 3 develops partial interpretation in theory. We anticipate its implementation, but we do not show much concern for practical considerations, especially performance. We treat the SMT solver as an oracle for our queries, within reason.

- Section 3.1 begins to move from complete information to partial information. We describe two representations for sets which resemble mathematical set comprehension. We show how to perform many operations on these sets.
- Section 3.2 shows how a program can be executed over partial information. This is straightforward (but elaborate) for straight-line code and conditionals. It is *not* straightforward for recursive calls, and we explain why not.
- Section 3.3 shows how we handle recursive calls, using a combination of techniques.
- Section 3.4 discusses the termination of partial interpretation, and the modifications this requires.

Chapter 4 follows through with our original plan: use the results of partial interpretation to perform specialization.

- Section 4.1 describes a specializer which is a client to partial interpretation. As expected, the specializer itself is relatively simple.
- Section 4.2 reports our experiments with our implementation. We reproduce basic results from partial evaluation, without manual adjustments to the input programs. We show improved accuracy over previous partial evaluators. However, the larger examples highlight the performance challenge with our "theoretical" implementation.
- Section 4.3 analyzes the performance of our implementation. We discuss several possible modifications to improve performance. One important theme is less reliance on the SMT solver. We achieve a speedup of approximately three orders of magnitude, compared to our "theoretical" implementation.

Chapter 5 concludes.

- Section 5.1 situates our work with respect to existing work. In particular, we highlight the ideas we have re-used from past work.
- Section 5.2 discusses what we have achieved in relation to our original goals.
- Section 5.3 reviews the limitations of our work, and discusses future work.

Throughout this text, we omit many implementation details as unessential. Our reference implementation can be consulted - see Appendix C for more information.

Chapter 2

Programs

We first introduce the program representation we will use as the object of our algorithms: ordinary interpretation, partial interpretation, and specialization. We choose to create a program representation rather than use an existing one. This allows us to adjust the program representation to the needs of partial interpretation; not the other way around. This is a deliberate decision, and this entire work can be seen as a feasability experiment for partial interpretation: let's see if it's possible, under the best conditions we can put together.

We design our representation according to the following ideals:

- express clean, well-understood programming constructs
- be well-suited for analysis by partial interpretation - in particular, be "close to SMT" whenever possible
- contain no superfluous information.

In addition to these ideals, sometimes we cut corners and come up short of a legitimate programming system. We do this in the practical interest of finishing the project, and seeing if partial interpretation can work. We will point out when we do this.

Our program representation expresses pure functional programs: in a call-return discipline, with no mutation or external effects. We feel that pure functional programs are a mature and well-understood subset of programming; whereas other features (non-local exits, mutation, and external effects) are less mature. We therefore leave these other features to future work.

We choose a graph representation of programs, that we refer to as a *pgraph*. Intuitively, nodes in a *pgraph* represent data and operations, and edges represent data dependencies. An *operation* → *data* edge indicates that the operation reads that data - this is in effect a "variable reference", abstracted from any naming problems. A *data* → *operation* edge indicates that the operation produces that data. Mnemonically, data nodes are red (boxes), and operation nodes are blue (ovals). This is illustrated in Figure 2.1. We clarify that this is an internal representation; not something a programmer would write directly.

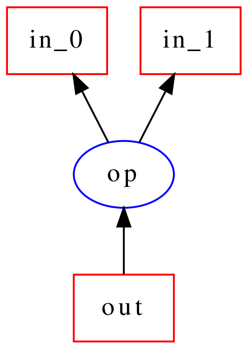

Fig. 2.1. A fragment of a *pgraph*: a non-descript operation, with two inputs and one output.

2.1. Node types

We introduce several node types that refine the intuitive "data and operation" nodes above. We introduce these in small, logically related groups.

2.1.1. Basic nodes: *result*, *entry*, *exit*

First, we introduce the node types *result*, *entry*, and *exit*.

A *pgraph* can have any number of *result* nodes, which are data nodes (red). A *result* has a single outgoing edge, to its producing operation. Operations can have several results, so the edge from a *result* to an operation is labelled with a result index.

Each *pgraph* has a single *entry* node, which is an operation node (blue). The *entry* has no outgoing edges, and represents the earliest moment in time in the execution of the *pgraph*. The results of *entry* are the formal parameters to the program.

Each *pgraph* has a single *exit* node, which is an operation node (blue). The *exit* has no incoming edges, and represents the latest moment in time in the execution of the *pgraph*. The data nodes *exit* depends on are the results returned by the program.

In our implementation, when an operation (such as *exit*) has several data dependencies, the edges representing them are ordered in an adjacency list. Mathematically, this ordering implicitly labels the edges outgoing from the operation.

With these node types, we can already assemble well-formed *pgraph*s: those that directly return some or all of their arguments. Figure 2.2 shows such a program, with the *result* label omitted.

2.1.2. Sub-programs: *prog*, *call*

Next, we introduce the node types *prog* and *call*, which are operation nodes (blue).

A *prog* node wraps a *pgraph*, and produces a single data value representing that *pgraph*. Complementarily, a *call* node represents the execution of a sub-program: the first outgoing

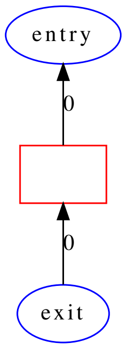

Fig. 2.2. A program which returns its single argument - i.e. the identity function.

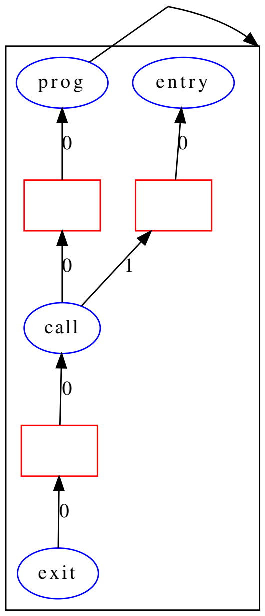

Fig. 2.3. A network containing a single *pgraph*, expressing the program `p(x) = p(x)`.

edge from *call* must be to a data value representing a *pgraph*; the other outgoing edges are the arguments to the sub-program.

In many ways, *prog* is similar to common programming notations whose names derive from mathematical functions: *func*, *fun*, *fn*, λ. Traditionally, these come with a notion of closure, whereas our *prog* node does not. We justify this limitation in two ways:

- partial interpretation is simplified by this limitation, and this is our main objective
- since we will have data structures (see 2.5), we can often recover the effect of closures, with a standard closure conversion or defunctionalization.[1]

A recursive program can be expressed as a network of *pgraph*s, connected through *prog* nodes, as in Figure 2.3. Visually, an arrow "connects" the *prog* node to the *pgraph* it wraps. However, this is not an *edge to a node*, and an individual *pgraph* contains no cycles.

[1]We make no precise claim here; we merely suggest that *pgraph*s are usable.

Primitive	**Type**
`false`	Bool
`true`	Bool
`not`	Bool → Bool
`and`	Bool * Bool → Bool
`or`	Bool * Bool → Bool
`+`	Int * Int → Int
`-`	Int * Int → Int
`*`	Int * Int → Int
`div`	Int * Int → Int * Int
`<`	Int * Int → Bool
`<=`	Int * Int → Bool
`>`	Int * Int → Bool
`>=`	Int * Int → Bool

Table 2.1. Boolean and integer primitives, and their types.

2.1.3. Primitive data and operations: *const*, *prim*

Next, we introduce the node types *const* and *prim*. The purpose of these is to inject primitive data and operations, into our programming system. Many choices could be made here, but we select a small set of primitives which maps well to SMT: boolean values, and the mathematical integers $\mathbb{Z}$. SMT solvers also have direct support for other primitive data types, including bit vectors (which can model machine words) and Unicode strings. These could be legitimate additions, but we do not explore them, in the interest of brevity.

A *const* node is a data node (red) which is annotated with a constant value, e.g. `true` or 3.

A *prim* node is an operation node (blue) which is annotated with a primitive operation symbol, e.g. `and` or `+`.

We use these to introduce the boolean and integer primitives shown in Table 2.1. Our division primitive produces both a quotient and a remainder. We will supplement these primitives with equality and data structures, in 2.5.

2.1.4. Conditional expression: *if*

Last, we introduce the node type *if*, which is an operation node (blue). An *if* node represents a conditional expression. Its first outgoing edge must be to a boolean value c. Two additional outgoing edges (the *consequent* and *alternate* branches) indicate which values are to be selected between, based on the value of c.

Our *if* node selects not only between values, but between execution paths. This is significant: though our representation is pure, there are inevitably operations that can produce errors (e.g. division by zero, or taking the head of an empty list). Additionally, recursion can be non-terminating. Our *if* node serves to guard the execution of such program elements.

We complete the definition of *pgraph*s by stating, in technical terms, that *if* is *non-strict* in its consequent and alternate branches; and that other *pgraph* elements are *strict*. Informally, this means that (except at *if*) every input to an operation is computed, before that operation begins executing. This is familiar from most programming languages.[2] At *if*, the condition is first computed; then, only one of the other branches (consequent or alternate) is computed.

2.2. Defining algorithms over *pgraph*s

We will define many algorithms over *pgraph*s: an ordinary interpreter, a partial interpreter, and a specializer. We start right away in the next section, with the ordinary interpreter.

We make a clarification here. When we say "an algorithm *over* a *pgraph*", we mean that a *pgraph* is the input *to* the algorithm; not that the algorithm is written *as* a *pgraph*. The algorithm is written in an existing implementation language, and the algorithm manipulates the input *pgraph* as data. No particular implementation language is essential to our presentation, but our implementation is written in Scheme.

There is a natural way to define an algorithm over a *pgraph*, and we will apply it repeatedly. The basic maneuver is to map each node n to an image $f(n)$. The nature of $f(n)$ depends on the particular algorithm. To compute $f(n)$, we allow ourselves to suppose that for all *successors* s of n (all s such that $n \to s$), $f(s)$ is available. In other words, we are defining the mapping f by structural induction. This induction is well-founded, since it is over an individual *pgraph*, which contains no cycles. Any circularity is between *pgraph*s, through *prog* nodes, and the induction never extends beyond a *prog* node.

Given the images of the immediate successors of n, and the local node and edge labels at n, we usually have enough to compute $f(n)$. Intuitively, this is because *pgraph* edges represent data dependencies in a program. This enables us to define algorithms over *pgraph*s, by specifying only their local behavior at each node type.

In our implementation, the mapping f is memoized: any image $f(n)$ is computed only once, and stored for later. In this text, we will rarely mention this memoization. Whenever a local function (at a node) accesses a successor image, this should be understood as memoized.

Overall, this maneuver amounts to a memoized depth-first traversal of a *pgraph*.

[2]Except notably Haskell, which is based on lazy evaluation.

2.3. Ordinary interpreter

We immediately apply the maneuver from 2.2, to define an ordinary interpreter for *pgraph*s. There is nothing complicated here, but we go through the motions for two reasons:

- to practice applying the maneuver, because we will later apply it to more complicated algorithms
- to see the ordinary interpreter, because our partial interpreter will derive from it.

We omit data structures, which will be introduced in 2.5 - handling them would add no value to this exercise. We therefore compute over the following data domain:

IVAL :=
  | `false` | `true`
  | INTEGER
  | PGRAPH

We map each data node to an IVAL; and each operation node to a list of IVALs, representing the (possibly multiple) results of that operation. We specify this mapping by giving its local behavior at each node type, as `interp-node`.

We define our interpreter as two mutually recursive functions:

- `interp-pgraph(pgraph, args)`
- `interp-node(node)`.

`interp-pgraph(pgraph, args)` simply invokes `interp-node` on the *exit* node of the given `pgraph`. It also makes `args` available to `interp-node`. This is a common pattern in our algorithms: the local function (here, `interp-node`) supposes access to certain free variables (here, `args`). We always explain where these free variables come from conceptually, but we omit the implementation details.

`interp-node(node)` examines the label of the `node`, and proceeds as follows.

- At *result*, fetch the list of IVALs at our single successor, and select one from it. The index of the one we want is the label on our single outgoing edge.
- At *entry*, return the `args` provided by `interp-pgraph`.
- At *exit*, fetch all the successor images, and package them into a list.
- At *prog*, package the wrapped *pgraph* into a singleton list.
- At *call*, fetch all the successor images. Then invoke `interp-pgraph`: the first successor image is the `pgraph`; the remaining images are the `args`.
- At *const*, the node annotation says which value to return.
- At *prim*, the node annotation says which primitive to apply. Fetch all the successor images, apply the primitive to them, and package the result(s) into a list.
- At *if*, first fetch the image of the condition. If it is `true`, package the consequent image into a list; if it is `false`, package the alternate image into a list.

2.4. Textual representation

The visual representation of *pgraph*s quickly becomes cumbersome, as programs grow. Later, we will want to discuss more substantial object programs. We therefore introduce a more compact textual representation, based on programming languages. Specifically, we use a surface syntax resembling a functional subset of Scheme [**AIBB$^+$98**]. Here is our recurring example of a factorial program, with the corresponding *pgraph* shown in Figure 2.4:

```
(define (fact n)
  (let ((t (fact (- n 1))))
    (if (<= n 1)
        1
        (* n t))))
```

The first line gives the name `fact` to the *pgraph* being defined. Beyond that, an expression in this syntax corresponds to a data node in the *pgraph*. The name `n` is given to the data node for the formal parameter. The `let` construct gives names to other data nodes. Here, the name `t` is given to the result of `fact(n-1)`.

This is mostly familiar, but there is one catch. A casual reading of the code above suggests that it does not terminate, because `fact(n-1)` is calculated before checking the condition. However, any intuition from programming languages should remain secondary to the primary purpose of this language: constructing *pgraph*s. The calculation `fact(n-1)` is *written* before the condition, but it is not *performed* before the condition.

The second line merely gives the name `t` to the data node for the result of `fact(n-1)`. That result is only *used* in the alternate branch of the conditional below. In the *pgraph*, this corresponds to the fact that the calculation `fact(n-1)` is guarded by the *if* node, and `fact(n-1)` will only be calculated when the condition is `false`. In other words, the following textual representation corresponds to the same *pgraph*:

```
(define (fact n)
  (if (<= n 1)
      1
      (let ((t (fact (- n 1))))
        (* n t))))
```

This factorial program terminates for all inputs, in all three forms. Note that our factorial is defined over all integers, with `fact(n) = 1` for negative `n`.

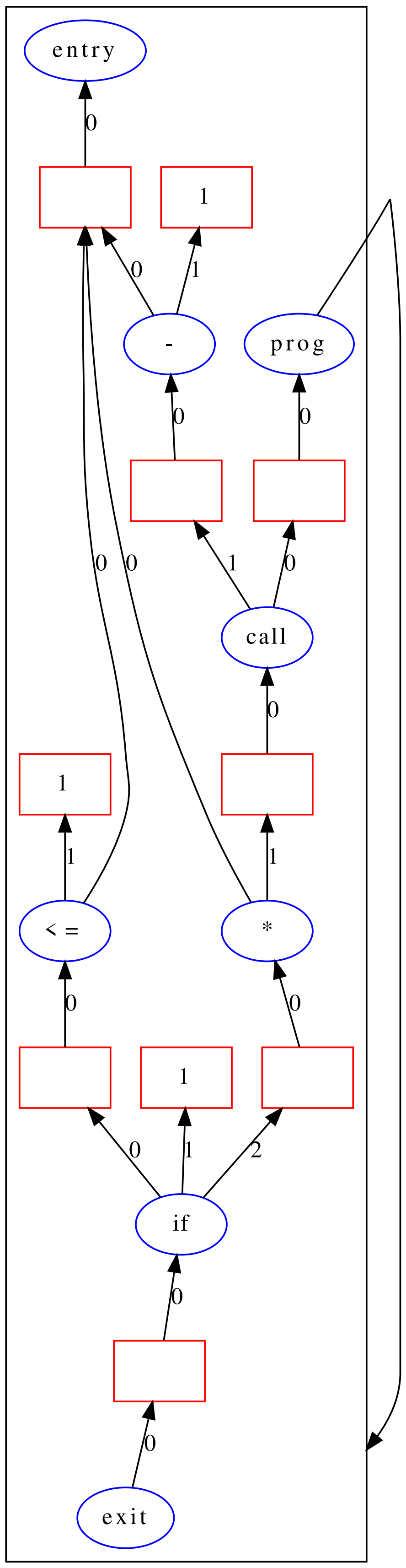

Fig. 2.4. A factorial program, exhibiting all major program elements.

Primitive	Type
`=`	α * $\alpha \to$ Bool
`mkPair`	α * $\beta \to$ Pair α β
`fst`	Pair α $\beta \to \alpha$
`snd`	Pair α $\beta \to \beta$
`nil`	List α
`cons`	α * (List α) $\to$ List α
`head`	List $\alpha \to \alpha$
`tail`	List $\alpha \to$ List α

Table 2.2. Polymorphic equality and data structure primitives, and their types.

2.5. Data structures and static types

Before we move on from programs, we want to add data structures as simply as possible. This is slightly complicated by our eventual use of an SMT solver, which requires that types be supplied for all symbolic values. In our experience, this is best achieved by having a static type system for our programs.

We introduce a standard system of static types, with parametric polymorphism, in the style of Hindley-Milner [**Mil78**]. This is the standard type system underlying statically typed functional programming languages, such as Haskell and OCaml. To simplify its treatment, we require that input and output types be supplied for each *pgraph*, by the programmer. We then perform a standard type verification[3], which ensures that the program is "well-typed". More importantly for us, as a byproduct we infer a type for each data node in the *pgraph*. These inferred types will be used during partial interpretation.

Under this kind of type system, there is a standard way to present general data structures: *algebraic data types*[4] (ADTs). In the interest of brevity, we do not provide full algebraic data types, which should include a mechanism for programmers to define their own data types. Instead, we provide built-in typed pairs and (homogeneously) typed lists, as a proof of concept. We leave a full implementation of ADTs to future work.

We introduce the polymorphic equality and data structure primitives shown in Table 2.2. These apply to data of any types α and β. `mkPair` creates a pair, whose components can then be accessed with `fst` and `snd`. `cons` creates a list, which can then be de-constructed as a `head` and a `tail`. `nil` expresses the empty list.

[3]We actually use an SMT solver to perform type verification, as an experiment. This works, but the only thing we appear to "gain" is a dependency on the SMT solver.

[4]For our purposes, algebraic data types are covered by [**Mil78**], though Milner does not use that name.

Chapter 3

Partial interpretation

3.1. Partial information

We now have programs (*pgraph*s, from Chapter 2), which are half of what we need before we present partial interpretation. We now discuss the other half: our partial information domain (sets of possible values). We plan to assign a set to each data node in a *pgraph*, during partial interpretation.

We build up to two set representations, and operations on them. We introduce several prerequisite ideas along the way, and we mention reasons for our choices. We clarify that our sets are not merely a notation on paper; they must become data structures in our algorithms.

3.1.1. Basic comprehension

Mathematical set notation is the starting point for our set representations. There are two broad approaches to writing a set in mathematics: in *extension*, where we enumerate the elements of the set; and in *comprehension*, where we write the *properties* of the elements of the set. Comprehension is the clear choice for us, because it includes the sets expressible in extension, and can additionally express infinite sets. We illustrate these observations shortly, as we examine set comprehension further.

The most basic form of mathematical set comprehension looks like this: $\{x \mid P(x)\}$. This notation designates the set of all objects x that satisfy a predicate P. Conceptually, there are two components to this notation: a name, and a proposition which may refer to that name. We examine these two components further.

First, in mathematics it is common to introduce the name x without specifying a type for x. In contrast, our set comprehensions will always come with a type for the elements of the set. We already know all these types. Recall that our plan is to assign a set to each data node in a program. But during type verification (see 2.5), we already inferred a type for each data node. Therefore it is easy to enhance our set comprehensions to introduce a *declaration* rather than just a name, in the following form: $\{x : \mathrm{T} \mid P(x)\}$. Here x is declared to have type T.

Second, in mathematics the proposition in set comprehension is written in (usually informal) "mathematical logic", and several logical constructs are accepted. For example, with equality and disjunction, we can embed extension in comprehension:

$$\{1, 2, 3\} \text{ (in extension)} \quad \text{becomes} \quad \{x : \mathbb{Z} \mid x = 1 \vee x = 2 \vee x = 3\} \text{ (in comprehension).}$$

With existential quantification and multiplication, we can express the even integers:

$$\{x : \mathbb{Z} \mid \exists k : \mathbb{Z}\ .\ x = 2k\}.$$

The *language* used to write these propositions affects which sets are expressible in comprehension. Conceptually, we would like to remain as close as possible to mathematical logic, which expresses mathematical sets. However, for the purpose of partial interpretation, we need some form of automatic deduction over these propositions. Practically, we use an SMT solver to provide this automatic deduction. Therefore, we conform ourselves to the language of propositions accepted by SMT solvers. Current SMT solvers give poor results when faced with recursive functions. Therefore, we do not use recursion in the propositions within our set comprehensions.

Note that we are not just trying to express sets, in a vacuum. We are trying to express those sets that will arise, when we propagate partial information through a program. Consider two sets which are the inputs to an addition operation. When we write the result of this operation, which is also a set, somewhere in the set comprehension we will need to write a $+$ operator. This $+$ operator will have SMT semantics, and it is simplest if those semantics match those of the + operator in our programs. This is why we have designed the basic elements of our programs to match SMT semantics.

This combined conformity to SMT limits the expressivity of our set representation, compared to mathematical logic. For example, without recursion or a primitive exponentiation operation, we cannot express the set of powers of two.

There is one more notion to introduce, before we arrive at our first major set representation. Our programs can receive several arguments (and return several results). We want to express *relational information* about these - for example, that one argument is greater than another. It is therefore not enough to describe a set, at each data node separately.

We introduce a notion of *arity* for our sets: we consider sets of possible values, not for a single data node, but for several at once. In other words, we consider sets of *tuples* of values; but these tuples are distinct from anything expressible as a data structure, in a *pgraph*. We make this choice so that the handling of relational information is dissociated from a particular system of data structures. An alternative could be to have a well-developed notion of tuple, in the system of data structures. A tuple could then package "multiple" arguments or results, as a single argument or result.

Consolidating all this, we come to our first major set representation, which we call *basic comprehension*. In our implementation, we dispense with the surface mathematical notation and store a pair `(decls, prop)`. Each declaration introduces a name with an explicitly given type. The number of declarations corresponds to the arity of the set - "how many" arguments or results it carries. The single proposition determines which combinations of values are members of the set. The declarations bind all the free variables in the proposition.

3.1.2. Extended comprehension

We could use basic comprehension as our sole representation for sets. In practice however, it is convenient to have two representations which are tuned to different situations. We reiterate that these set representations correspond to concrete data structures, whose details we omit here; our written notations merely suggest the contents of these data structures. Consider again the example of an addition operation with our first set representation, this time in more detail. Suppose we want to add $\{x : \text{Int} \mid P(x)\}$ and $\{y : \text{Int} \mid Q(y)\}$. Clearly the result "is":

$$\{x + y \mid \exists x : \text{Int}, y : \text{Int} \ . \ P(x) \wedge Q(y)\},$$

but we would have no way of writing this directly. In basic comprehension, the left-hand side must be a declaration - not an expression.

We therefore introduce a second set representation which we call *extended comprehension*, and which corresponds to standard set builder notation. The representation becomes `(exprs, decls, props)` - note the plural `props`. Each expression may refer to the declared names. The number of expressions corresponds to the arity of the set. The declarations are implicitly existentially quantified, and the propositions are implicitly conjoined. Again, the declarations bind all the free variables in the propositions (and in the expressions).

With this second representation, operations on values are easily lifted to operations on sets of values. Suppose we want to perform a binary operation (such as addition) on two sets of arity 1, represented as (e_1, ds_1, ps_1) and (e_2, ds_2, ps_2):

$$(e_1, ds_1, ps_1) + (e_2, ds_2, ps_2) = (e_1 + e_2, ds_1 \cup ds_2, ps_1 \cup ps_2)$$

Here we extend the notation to let e_1 and e_2 refer to individual expressions. In 3.2, we will discuss how to handle the (here informal) unions - of the declarations, and of the propositions.

We will use extended comprehension extensively during partial interpretation, within a given *pgraph*, where primitive operations are pervasive. We will also use the first representation (basic comprehension: `(decls, prop)`) because it is preferable when defining several set operations. We will see why we need these set operations, in the next sections.

3.1.3. Conversion between the two representations

Recall:

- basic comprehension: `(decls, prop)`
- extended comprehension: `(exprs, decls, props)`

These two representations are mathematically equivalent: we can convert between them, and they can express the same sets.

Given a set in basic comprehension `(decls, prop)`, it is trivial to construct an extended comprehension, as follows. To construct `exprs`, reference each name in `decls`. The new `decls` are the same as the `decls` in basic comprehension. Finally, the `props` contain the single `prop` from basic comprehension. For example:

`(x:Int, x > 0)` becomes `(x, x:Int, x > 0)`.

Given a set in extended comprehension `(exprs, decls, props)`, with a little work we can construct a basic comprehension, as follows. Conceptually, we introduce a fresh declaration for each expression, and these fresh declarations become the `decls` of the basic comprehension. The types for these declarations are once again available to us, because they correspond to data nodes in a *pgraph*.

It remains to construct the proposition for basic comprehension, and it involves three pieces. First, we explicitly conjoin the propositions from extended comprehension. Second, we additionally conjoin the assertions that the new names are equal to the expressions from extended comprehension. Third, we explicitly quantify the *old* declarations - this makes them "disappear" into the proposition.

We summarize this symbolically. Given an extended comprehension `(exprs, ds, ps)`, we construct the basic comprehension `(decls, prop)` as follows:

- `decls` $=$ a sequence of fresh names `xs`, with the same types as the `exprs`
- `prop` $=$ $\exists$`ds` . ((`xs` $=$ `exprs`) $\wedge$ `ps`)

We omit the details of quantifying over several declarations `ds`, the fact that `xs` $=$ `exprs` is really a sequence of equalities, and the explicit conjoining of the original propositions `ps`.

For example:

`(x-1, x:Int, x > 0)` becomes `(x:Int, ∃k:Int . x = k-1 ∧ k > 0)`.

Renaming the explicitly quantified variable (here, to k) preserves a clean naming scheme for free variables, and facilitates substitution in the implementation.

3.1.4. Set operations

We end this section with a discussion of set operations. We will use these set operations in the formulation of partial interpretation, in the next sections. We discuss the following set operations: complement, intersection, union, emptiness, uniqueness, inclusion, and equality. We define these on basic comprehension.

The first three operations can be implemented by syntactic manipulation alone. Set complement keeps the same declarations, and takes the negation of the proposition.

Set intersection is defined on two sets (ds_1, p_1) and (ds_2, p_2), only when the types of ds_1 and ds_2 match. The result is $(ds_1, p_1 \wedge q)$, where q is obtained by substituting from (the names in) ds_2 to ds_1, in p_2. This assumes that there are no name clashes in this substitution.

Set union is the same as intersection, except that the proposition is $p_1 \vee q$ instead of $p_1 \wedge q$. For example, $\{x : \text{Int} \mid x < 0\} \cup \{y : \text{Int} \mid y > 0\} = \{x : \text{Int} \mid x < 0 \vee x > 0\}$.

The remaining operations make use of an SMT solver. In our implementation, we use the textual language SMT-LIB [**BST$^+$10**] to interact with the CVC4 solver [**BCD$^+$11**].

The emptiness check is a key operation. We first declare all the `decls` into a fresh SMT context. Then, the `prop` is asserted. Finally, a CHECK-SAT command is issued, and the SMT solver reports either SAT(isfiable), UNSAT(isfiable), or UNKNOWN. If the result is:

- SAT, then the set is inhabited
- UNSAT, then the set is empty
- UNKNOWN, then it is not known whether the set is inhabited or empty.

We give a brief example. Consider the set

$$\{(x : \text{Int}, y : \text{Int}) \mid x < y\}.$$

To check its emptiness, we issue the following SMT commands:

```
(declare-const x Int)
(declare-const y Int)
(assert (< x y))
(check-sat)
```

The SMT solver reports SAT, and we know this set is inhabited. This interaction (and others) can be enclosed between PUSH and POP commands, to keep the SMT context clean for further queries.

The uniqueness check begins like an emptiness check: we declare the names, assert the proposition, and check for satisfiability. At this point, it makes sense to continue only if the set is inhabited. We then use another SMT solver feature: whenever the SMT solver reports SAT, we can further ask it to exhibit a *model*. This is an assignment of values to all the variables, which satisfies all the constraints. We now assert that at least one of the variables is *distinct* from this model, and we check for satisfiability again. If the result is:

- SAT, then we have exhibited two points within the set - they are therefore *not* unique;
- UNSAT, then the original model was the only point within the set - it is unique;
- UNKNOWN, then it is not known whether the set contains a second point.

We give a brief example. Consider the set

$$\{x : \text{Int} \mid 0 < x < 4 \land \exists k : \text{Int} \ . \ x = k^2\}.$$

This is the set of squares, constrained to lie between 0 and 4 (exclusively).

```
(declare-const x Int)
(assert (and (< 0 x) (< x 4) (exists ((k Int)) (= x (* k k)))))
(check-sat)
(get-model)
```

The solver reports SAT, then produces the model $x = 1$. We now continue.

```
(assert (distinct x 1))
(check-sat)
```

The solver reports UNSAT, and we know the original set has the unique member 1.

Set inclusion can be formulated in terms of the operations above:

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

Set inclusion can report UNKNOWN, since it relies on an emptiness check.

Set equality can be formulated as double inclusion:

$$S = T \iff S \subseteq T \land T \subseteq S$$

The conjunction used in set equality must account for the inclusions reporting UNKNOWN. For this purpose, we define a ternary conjunction, as follows:

$a \land b =$

| if either input is definitely `false`, then the result is definitely `false`
| otherwise, if either input is `unknown`, then the result is `unknown`
| otherwise, both inputs must be definitely `true`, and the result is definitely `true`.

3.2. Basic partial interpretation

We now have programs (*pgraph*s, from Chapter 2), and a notion of partial information as a set of possible values (from 3.1). We are ready to present partial interpretation: computation over partial information.

Our presentation conceptually starts from our ordinary interpreter (from 2.3). We then modify the interpreter progressively, to handle partial information at the eight *pgraph* node types (from 2.1): *result*, *entry*, *exit*; *prog*, *call*; *const*, *prim*; *if*.

In our development of partial interpretation, sometimes we cut corners in the interest of brevity - as with programs in Chapter 2. We will point out when we do this.

3.2.1. Extended comprehensions: *result*

In the ordinary interpreter, given a *pgraph* and some arguments, we assigned an ordinary value to each data node.[1] In the partial interpreter, we now assign an extended comprehension (of arity 1) to each data node.[2] This looks like `(expr, decls, props)`.

In the ordinary interpreter, we also assigned a *list* of values to each *operation* node - these were the (possibly multiple) results of the operation. By analogy, we should now assign a list of extended comprehensions (each of arity 1) to each operation node.

However, with partial information there can be relational information between multiple results. To express this relational information, we make the results share a single group of `decls` and `props`. We then consolidate all the results as a single extended comprehension (of arity equal to the number of results). This looks like `(exprs, decls, props)`.

From this correspondence between data nodes and operation nodes, we directly get the behavior of the *result* node during partial interpretation. All we need to do is select one of the expressions from our producing operation, and reproduce the `decls` and `props`.

3.2.2. Straight-line code: *const*, *entry*, *prim*

We now discuss basic *pgraph* elements: constants, formal parameters, and primitive calculations over these.

At *const*, it suffices to construct the SMT expression representing the wrapped value. Since that value is a constant, the expression does not refer to any variables, and no `decls` or `props` are needed to "support" the expression. We postpone the discussion of expressions until we interact with an SMT solver (in 3.2.7).

[1]Because *if* is non-strict, we didn't necessarily assign a value to *every* node. We ignore this detail here.

[2]In the context of program analysis and optimization, it is somewhat unusual that we assign anew to each node, given a *pgraph* and some arguments. In technical terms, partial interpretation is (highly) *polyvariant*.

At *entry*, we suppose that our formals are given to us in extended comprehension. They originate from our caller, and they specify the set of possible inputs we are to compute with. In general, they come with `decls` and `props`.

At *prim*, we use our idea from 3.1.2: we can lift an operation on values, to an operation on extended comprehensions. Recall the formula for lifting a binary operation:

$$(e_1, ds_1, ps_1)\ op\ (e_2, ds_2, ps_2) = (e_1\ op\ e_2, ds_1 \cup ds_2, ps_1 \cup ps_2)$$

Here e_1 and e_2 are single expressions, which is what a binary primitive operation will find at the data nodes it depends on - in addition to the declarations ds_1 and ds_2, and the propositions ps_1 and ps_2. We can suppose that these results are available at our successor nodes, by structural induction - just as in the ordinary interpreter.

This lifting formula applies similarly to primitives of different arities - in particular to unary operations, and to division (which produces two results). The point is that the result expression(s) are constructed by syntactic manipulation of the input expression(s). The `decls` and `props` (from all the inputs) are combined. Combining the `decls` and `props` raises some subtle points, which we discuss next.

3.2.3. Set vs computational history

Suppose we are given a completely unknown integer x, and we compute both $2x + 1$ and $2x - 1$. We get two extended comprehensions: `(2x+1, x:Int, ∅)` and `(2x-1, x:Int, ∅)`. Now suppose we subtract them. Mechanically, there is a question of how to combine their `decls`, because each input declares `x:Int`.

Intuitively, we know that the two declarations should be merged into a single declaration, and that the difference is 2. But if we view $2x + 1$ and $2x - 1$ as *sets*, then they should be self-contained. A set should declare its own variables, and we should rename one `x` before combining the `decls`. If we do this, we get e.g. `(2x+1 - (2y-1), x:Int y:Int, ∅)`. And if we view *this* as a set, it is not $\{2\}$; it is the set of even integers.

This is not a mere mechanical question; it is a conceptual problem, and we approach it as such.

Clearly we have *lost something* by viewing $2x+1$ and $2x-1$ as sets. Intuitively, we have lost the knowledge that they were computed from *the same x*. Another intuition is that we know $2x + 1$ and $2x - 1$ are distinct, partially known *values*. But if we view them as *sets*, they are *equal*: the sets $\{2x + 1 \mid x : \text{Int}\}$ and $\{2x - 1 \mid x : \text{Int}\}$ have the same members - the odd integers.

This shows that sets are not the only notion of partial information. We are dealing with a *second*, distinct notion, which we have left intuitive so far. We propose to think of this second notion as a *computational history*: a "log" of how a piece of data has been produced.

When computing over complete information, we are not used to caring much about computational history. If we are computing the result of $2 + 3$, we report 5. It is almost considered a *mistake* to report $2 + 3$ instead. 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 when we discard this information, because it 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 rather than $2 + 3$.[3] This justifies the intuition that we should report 5 rather than $2 + 3$.[4]

When computing over *partial* information, history can matter. We have seen this in the example of $2x + 1$ and $2x - 1$. In general, the most assuredly accurate form of partial information must be *complete computational history*. Still, we have already seen one case when discarding history does not lose accuracy: complete information. We will discuss other such cases later, in 3.3.

For now, we note that since we are aiming for complete accuracy, in general we should be computing histories rather than sets. We first need to represent histories somehow, and we can use essentially the *same representation* as for sets: extended comprehension. The only difference is the *scoping* of the declarations and propositions.

When we viewed $2x + 1$ and $2x - 1$ as sets, they each declared a distinct x. To represent their histories properly, we need to declare x once for both. To achieve this, we introduce a *provenance* for `decls` and `props`: the node where they were introduced, in the *pgraph*. Right now they are only introduced at *entry*. This will remain their main provenance, but later we will see that some other nodes can also introduce `decls` and `props`.

The *scope* of a node n (which introduces `decls` or `props`) is the set of nodes which have a path to n. In the other direction, each node uses the `decls` and `props` introduced at its (transitive) dependencies. This settles the question of how to combine `decls` and `props` (such as at a binary *prim* node): merge them if they have the same provenance; otherwise keep them distinct.

Although we are now using computational history as our main notion of partial information, sets remain relevant for us. One reason is that eventually we will want to check whether the result of an operation is unique - to remove that operation from the specialized program. To check this, we will convert that result to a set, then check uniqueness as in 3.1.4.

To convert from a history to a set, conceptually we forget the history, and we keep only the possible end results that can come out of that history. Concretely, we forget the provenances of the `decls` and `props`. The resulting history becomes self-contained: it declares its own variables. In other words, it has no common past with any other history.

[3]Assuming the same resources are needed to store 2, 3, or 5; not to mention the $+$.

[4]We note that debugging could be greatly facilitated by knowing that something is $2 + 3$ rather than 5.

We emphasize a distinction here, between the notion of set and our implementation of it. Conceptually, a set does not have a history. Concretely however, we represent a set as a select history, which *could* have produced the set's members (e.g. $2x+1$ rather than $2x-1$). Therefore forgetting provenance is not enough to turn a history into a set. We must also *treat* it as a set in subsequent operations (e.g. checking set equality).

3.2.4. *prog*, *exit*

We have seen the basic behavior of partial interpretation: values are histories represented as extended comprehensions. We combine their expressions syntactically, and we combine their `decls` and `props` by tracking their provenance. This corresponds to *result*, *const*, *entry*, and *prim*. We now briefly discuss *prog* and *exit*, before moving on to *if* and *call*.

prog is similar to *const*: it produces an expression (representing the wrapped *pgraph*), without needing any `decls` or `props` in support. Each source *pgraph* is identified by an integer. This identification is established before beginning partial interpretation, and stored in a table. During partial interpretation, the correct SMT expression is constructed by looking up this integer, and wrapping it in a `mkProg` constructor. Again, we postpone the discussion of expressions until we interact with an SMT solver (in 3.2.7).

At *exit*, we report the history we have computed for the results. This history can depend on our formal parameters, and it is important that we report these dependencies to our caller. Concretely, at *entry* our formals contained `decls`. We can use the same names in the results, to indicate dependencies on the inputs.

To see the significance of this, consider a small program `p(x) = x - id(x)`, where `id` is the identity function, and `x` is a completely unknown integer. Disregarding the details of making a call for now, `p` calls `id` with a completely unknown integer argument. If we are not careful, `id` then returns "a completely unknown integer".

It is *true* that the result of `id` is a completely unknown integer; but this is not the whole truth. It is more accurate to say that the result is the *same* completely unknown integer as the input to `id`. This enables `p` to reason that the final result is 0.

3.2.5. Simple but non-terminating *if* and *call*

Summarizing so far, we have examined six node types: *result*, *const*, *entry*, *prim*, *prog*, and *exit*. Handling these has made the partial interpreter more complex than the ordinary interpreter. Nevertheless, we have made no real *concessions*: over these node types, we have not sacrificed any accuracy in the partial information computed.

We are about to examine the remaining node types: *if* and *call*. When we do this, we will inevitably encounter the main difficulty with partial interpretation: termination of the partial interpreter over recursive object programs. We first describe simple (but non-terminating) behaviors for *if* and *call*. We use these simple behaviors to illustrate the termination problems that arise in partial interpretation. We remedy part of the problem by modifying the handling of *if*. This adds more complexity to the partial interpreter: interaction with an SMT solver. Despite this, some termination problems remain. We devote 3.3 and 3.4 to addressing these problems, by modifying the handling of *call*. We will see that achieving complete accuracy there is a substantial problem, with many opportunities for future work.

To handle the *if* node, one simple idea is to handle it like a primitive operation. To re-iterate, this idea fails, but we describe it to illustrate the problems we run into. The language of SMT expressions includes a conditional expression, written `ite` - as in if-then-else. It is therefore possible to write an expression representing the result of the *if* node. First compute all the data dependencies: the condition c, the consequent value a, and the alternate value b. The result expression is then (`ite` c a b). Intuitively, this is equivalent to:

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

Here we are making no attempt to select between the two branches; we are merely encoding the selection into a logical proposition.

To handle the *call* node, the simplest idea is to handle it as in the ordinary interpreter: a *call* in the object program is mirrored by a (recursive) call to the interpreter. The actual arguments directly become the formal parameters for the called program - full history included. We have not implemented this, and we omit most of the details.

We detail one point, which remains relevant in a terminating implementation of *call*: we need to determine which program to call. What we are given is a set of possible programs to call. We resolve this in a simple way, in this work. We check if this set has a unique member (see 3.1.4). If it does, we know which program to call. If not, we report a completely unknown result from the *call*. Such an example - a call to a non-unique program - might arise in a root call to `map`, where the function argument is unknown.

There is an opportunity for additional accuracy here, but we do not pursue it, in the interest of brevity. When the set of possible programs is not a singleton, we could attempt to enumerate its members. We could then execute each possible program, and combine the results - either in a union, or in a conditional expression which records which program produced which results. Enumerating members is possible if the set is known to be finite. This guarantee depends on the details of handling cross-module calls, which we do not investigate.

The enumeration itself is an extension of the uniqueness check (see 3.1.4). Recall that the uniqueness check starts by obtaining one possible value; then, we reject this value and check for satisfiability again; if the value is not unique, we get SAT. To enumerate the set's members, we can continue by obtaining a *second* possible value. Then, we reject this value and check for satisfiability again - etc.

Combined with the previous node types, these simple behaviors for *if* and *call* ensure that partial interpretation terminates on object programs without recursion. We are effectively encoding every possible computational path, into a large set comprehension. However, this does not terminate on object programs with recursion.

3.2.6. Non-termination of recursion

Consider the partial interpretation of a factorial program, on completely known input 1. The *if* node makes no attempt to decide the branch: we have stipulated that it should merely construct a conditional expression. Accordingly, the *if* node recursively requests the condition, and the consequent and alternate branches, intending to package these into a conditional expression. The alternate branch is a *call* of `fact(0)`, which is handled by a recursive call to the partial interpreter. In the partial interpretation of `fact(0)`, the same thing happens, and we have an infinite loop in the partial interpreter.

This is the same problem as in the ordinary interpreter: the *if* node should select not only between values, but between execution paths. The solution is also the same: we should attempt to decide the condition. When we succeed, we should request only one of the consequent or alternate branches.

Therefore, we want to decide branches whenever possible. This is not quite as critical as the above example suggests: the simple handling of *call* amplifies the problem, and that is not the handling of *call* we will use ultimately. However, the principle remains: even with our handling of *call* in 3.3, it will be good to decide branches whenever possible.

3.2.7. Handling *if* with an SMT solver

The plan is to use an SMT solver to decide branches. We will carry an SMT process through partial interpretation. We first describe the initialization of this SMT process. We then discuss how to modify the behavior of the previous node types, to interact with the SMT solver. Then, we give the modified behavior of the *if* node.

The partial interpretation of each *pgraph* begins with a fresh SMT process. We first issue the following datatype declarations:

```
(declare-datatype Prog ((mkProg (progId Int))))
(declare-datatype Pair (par (T U) ((mkPair (fst T) (snd U)))))
(declare-datatype List (par (T) ((nil) (cons (head T) (tail (List T))))))
```

These declarations augment the language of expressions accepted by the SMT solver, beyond those natively supported by the SMT-LIB language. Since we selected our primitive data and operations to closely match SMT semantics, the basic SMT-LIB language already provides everything else that we need: booleans, integers, and operations over these.

The Prog type enables us to write the SMT expression `(mkProg _)`, which expresses the output of a *prog* node: a *pgraph* represented by an integer.

The parametric type families Pair and List enable us to write the SMT expressions `(mkPair _ _)`, `nil`, and `(cons _ _)`, which express pairs and lists.

We could leave the behavior of previous node types unchanged. This would mean that any time we wish to make an SMT query (such as at an *if* node), we would pipe all the `decls` and `props` reaching that node, into the SMT solver. After making our query, we could reset the SMT solver.

Instead of doing this, as we traverse the *pgraph* during partial interpretation, we incrementally update a single SMT context. Whenever `decls` and `props` are introduced, we immediately pipe them to the SMT solver, in addition to writing them as the result of partial interpretation at that node. Our memoized traversal of the *pgraph* ensures that this is done only once at each node, which avoids duplicate declarations to the SMT solver.

We now describe the modified behavior of the *if* node. We first obtain the result of the condition. This contains an expression, and we query the SMT solver to check if this expression has a unique value. The uniqueness check is described in 3.1.4, but here it is simpler: the names are already declared, and the propositions are already asserted. We introduce a temporary name c for the condition, to ease the interaction:

```
(push)
(declare-const c Bool)
(assert (= c ...))
(check-sat) ;; we know this is 'sat'
(get-model)
```

The SMT solver reports a possible value for the condition c, which is a boolean. We now assert that the condition c is distinct from this possible value, and CHECK-SAT again. We enclose this entire interaction in a PUSH-POP pair, to restore our SMT context after this query. We do not wish to definitively assert that the condition has a certain value. Intuitively, POP returns the SMT solver to its state before the last PUSH, undoing everything in between.

```
(assert (distinct c ...))
(check-sat)
(pop)
```

The SMT solver reports one of three things:

- SAT, in which case the condition of the *if* is unknown - it has two possible values, which is all the possible values for a boolean
- UNSAT, in which case the condition of the *if* is known, and equal to the first possible value reported by SMT
- UNKNOWN, in which case we must be conservative and suppose that both boolean values are possible - throughout our algorithms, it is straightforward to be conservative in handling UNKNOWN.

When the condition is uniquely known, we obtain the result of the corresponding branch (consequent or alternate), and we report it directly as the result of the *if* node. When the condition is unknown, we fall back on our earlier implementation of *if*: we obtain the result of both branches (consequent and alternate), and combine them into a conditional expression.

There is an additional opportunity for accuracy here, but we do not pursue it, in the interest of brevity. When the condition is unknown, we could compute the result of the consequent branch *under the assumption that the condition is true* - and conversely in the alternate branch. This is sometimes referred to as tracking a *path condition*, in the literature on *symbolic execution*. This would complicate our algorithm, because a node's value could depend on which path we came from.

Note that other than the limitation we just described, this handling of *if* makes no sacrifice in accuracy, so long as the SMT solver does not report UNKNOWN.

Giving an example of SMT reporting UNKNOWN is tricky for two reasons. First, SMT solvers are the subject of active research, and they are constantly improving their methods. Second, when UNKNOWN arises in our usage, it is typically for a very large query. Note that when UNKNOWN arises in our usage, we have observed empirically that specialization accuracy is not negatively affected. In other words, when SMT couldn't find an answer, there was no useful answer to be found (in terms of further specializing our programs).

We give a small synthetic example of UNKNOWN which we encountered early on: is it true that for all lists of integers x, there exists a list y such that x is the `tail` of y? Intuitively we know this is true, because we can exhibit y by `cons`ing absolutely anything in front of x. CVC4 reported UNKNOWN on this query. We reported this to the CVC4 mailing list, it was fixed, and our example was added to the CVC4 test suite. However, if x is a list of lists of integers, CVC4 still reports UNKNOWN. We reported this as well, but we have not heard back. To be fair, we did not attempt to learn and fix their code ourselves either.

This example reflects the fact that SMT solvers traditionally excel at proving existentials (i.e. finding a concrete model), but not universals (i.e. finding a symbolic proof). Note that universals occur naturally as negated existentials. The weakness with universals is being mitigated by recent work, e.g. *counterexample-guided quantifier instantiation* [**RKK17**].

3.2.8. Remaining non-termination

Despite the modified implementation of *if*, there are still termination problems. Consider the partial interpretation of `fact(x)` - factorial of a completely unknown input `x`. The details of this example depend on whether or not we track path conditions (see 3.2.7), but the end result is the same regardless. We examine both cases, to see that the result is the same.

Suppose first that we do not track path conditions - we do not track them, in this work. The *if* node requests the condition and attempts to decide the branch, but is unable to. Therefore, it requests both the consequent and alternate branches - intending to package them into a conditional expression. The alternate branch is a *call* of `fact(x-1)`, which is still a completely unknown integer, and we have an infinite loop in the partial interpreter.

Suppose next that we do track path conditions. Again, the *if* node fails to decide the branch. Then, when requesting the alternate branch `fact(x-1)`, it does so under the assumption that $x \geq 2$. Therefore the first recursive call has argument:

$$\{x - 1 \mid x : \text{Int} \; . \; x \geq 2\}.$$

The second recursive call will have argument:

$$\{x - 2 \mid x : \text{Int} \; . \; x \geq 3\}.$$

And again we have an infinite loop in the partial interpreter.

This illustrates a fundamental problem: **it is possible for a program that terminates on all (completely known) inputs, to not terminate on partially known inputs**. One intuition is that during ordinary execution, every branch is successfully decided, and we go into a single path of execution. During partial interpretation, whenever a branch is *not* successfully decided, we go into *two* paths of execution. Therefore we are executing *more* code than during ordinary execution. Another intuition is that termination depends on going into a "base case", which eventually happens for all completely known inputs.[5] However as we see here, under partial information it is possible to never definitively go into the base case.

To address this, we must abandon the simple implementation of *call*, which mirrors a call in the object program by a recursive call to the interpreter.

[5]For an ordinarily terminating program (such as factorial).

3.3. Handling recursion

We have just seen how to perform partial interpretation, at every node type except *call*. We have also seen that partial interpretation of `fact(x)` - factorial of a completely unknown integer `x` - does not terminate, under the simple handling of *call* described in 3.2.5.

In this section, we show how to modify partial interpretation so that it terminates in the presence of recursive calls. We introduce many modifications: not only do we modify the handling of *call* nodes; we also modify the overall structure of partial interpretation.

We begin by making a key observation about the example of `fact(x)`. This time we omit discussing the nuances of path conditions (see 3.2.7 and 3.2.8); as in 3.2.8, the end result is the same regardless. We already hinted at this key observation, in 3.2.8. We justified our statement that partial interpretation of `fact(x)` does not terminate, as follows: `x` is completely unknown, and in the recursive call, `x-1` is *still completely unknown*. In some sense, there is repetition from one call to the next.

Taken as *values*, the arguments to two successive calls are obviously distinct: the second is one less than the first. But taken as *sets*, they are *equal*. The sets $\{x \mid x : \text{Int}\}$ and $\{x - 1 \mid x : \text{Int}\}$ contain the same members - each set is included in the other.[6]

We can leverage this repetition to achieve termination of partial interpretation. Our central point is that if the *arguments* are equal (as sets), then the *results* are equal (in some sense which we will clarify). In other words, we are suggesting that we can discard all history from the arguments to calls, to turn them into sets - at no loss in accuracy. We first explain why we consider this justified. The rest of this chapter shows how it gets us termination.

[6]Recall that our definition of factorial (see 2.4) is over all integers, with `fact(x)` $= 1$ for negative `x`.

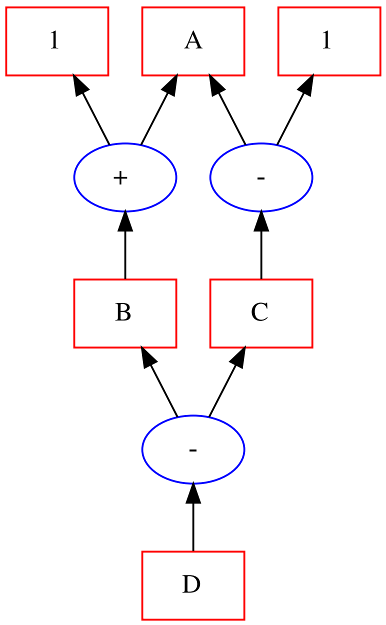

Fig. 3.1. Diamond-shaped *pgraph* fragment, illustrating the example from 3.2.3.

3.3.1. Arguments as sets

We have seen that in general, discarding history can lose accuracy. But we have also seen that the history of a completely known value can be discarded, without losing accuracy. We now elicit other conditions under which history can be safely discarded.

Consider Figure 3.1, which illustrates our example of $2x + 1$ and $2x - 1$, from 3.2.3:

A: $2x$ B: $2x + 1$ C: $2x - 1$ D: $2x+1-(2x-1)$

We saw that in this example, history mattered. Specifically, it was necessary to store the histories of B and C - up to A, at least. This enabled us to accurately compute D as 2.

Abstractly, the first feature of this example is that B and C share a common past at A. If they do not share a common past anywhere, knowing their history gains us nothing.

The second feature is that (the producing operation of) D later *rejoins* B and C. This is when their shared history actually matters. Before this, the shared history is being passively preserved. If B and C are never rejoined, the shared history is preserved for nothing.

The third feature is that there is no path between B and C. This is true in both directions, and this symmetry is complicating our analysis. We focus our attention on B. We conjecture the following necessary condition for the relevance of the history of B up to A:

There exists D with D $\rightarrow^*$ B $\rightarrow^*$ A, and D also has a *side path* to A, not through B.

This condition is depicted in Figure 3.2.

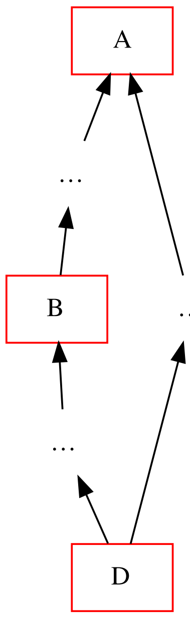

Fig. 3.2. Conjectured necessary condition for the relevance of the history of B up to A.

We do not know how to prove this, or how to define the (ir)relevance of a history. Intuitively though, suppose D (transitively) depends on A *only through B*. Then in any expression for D, we can *eliminate* A by rewriting in terms of B. It seems then that any relational information about A is captured in the relational information about B.

In the concrete example, $2x+1$ and $2x-1$ depended on x, but only through $2x$. We can therefore set $u = 2x$, and reason about the final result as $u+1-(u-1)$. We get the same conclusion, regardless of the history of u in terms of x - only the history up to u matters.

Supposing this conjecture is correct, we can justify discarding all history in the arguments to any call. In Figure 3.2, suppose B represents the arguments to a call. A is any piece of the history of the arguments, in the caller. D is anything computed within the call. D *cannot have a side path* to A, because a *pgraph* can only receive information through its arguments: there are no closures or global variables. It would also make no sense to access data directly in the caller's scope, because in principle we don't even know what program made the call.

Once the history of the arguments is discarded, we view them as a set (of tuples), which must determine the result of the called program. As discussed in 3.2.4, the result is a history which depends on the formal parameters. In other words, the result (as an input-output relation) is completely determined by the arguments (as a set). The caller can then replace the formals with the arguments (in the result), to recover the history in the caller's scope.

We give a brief example: `p(x) = succ(x) - succ(x-1)`, with `succ(y) = y+1`, and `x` completely unknown. Here the arguments to `succ` are `x` and `x-1`, which are equal as sets. We can execute `succ` a single time, with a formal parameter of `y`, and the result `y+1`. Then for the first call, `p` substitutes `y` $\rightarrow$ `x` in the result; for the second call, `y` $\rightarrow$ `x-1`.

3.3.2. Caching results

These observations justify our first modification to the handling of *call*. We introduce a global cache that maps a pair `(pgraph, args)` to a result. The `args` are treated as a set, and the result as an input-output relation. The idea is that when we reach a *call* node, we attempt to avoid re-executing the sub-program. We first look up the relevant pair `(pgraph, args)`, and attempt to re-use a stored result.

Note that the *possibility* of such a cache depends critically on two aspects of our development. First, we need *something to store in the cache.* This is where a call-return discipline is especially beneficial: when we go into a call, there is a well-defined future point when we will return. The cache stores the results at that point. Everything that would have happened - between the call and the return - can be discarded if we have the results. Intuitively, a cache hit then allows us to "fast forward" through the execution of the call.

Second, we need a notion of equivalence of arguments, to decide what constitutes a cache hit. In this work, we use set equality as the notion of equivalence. We implement this by converting the arguments to basic comprehension (see 3.1.3) at the beginning of a *call*. Then they are compared for equality (see 3.1.4), to the arguments in existing cache entries. If there is no cache hit, we actually execute the call. At this point, the fresh variables of basic comprehension neatly separate the actual arguments from the formal parameters. All *entry* has to do is convert back to extended comprehension.

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

3.3.3. Finding a fixed point

By itself, the addition of this cache does not solve the termination problem. Consider again the partial interpretation of `fact(x)`, as it stands. We make a root call, planning to later save the result in the cache. When we reach the first recursive sub-call, we consult the cache but find nothing. We therefore recursively execute the sub-call, planning to later save the result in the cache - etc. We never get a result, we never store anything in the cache, and we never terminate.

The problem is a circular dependency: we need the result of `fact(x)` to compute the result of `fact(x)`. We can express this with a recursive equation:

$$R = F(R)$$

Here R denotes the result of `fact(x)`, and F represents part of the *pgraph* of `fact`: the data path from the return point of the *inner* `fact`, to the return point of the *outer* `fact`.

Restating this equation in words, R is a fixed point of F. We can therefore hope to solve for R with an iterative fixed point computation. That is our plan, and the rest of this chapter discusses the implications of this plan.

Intuitively, we will compute a sequence of tentative results $R_0, R_1, ...$, with $R_i = F(R_{i-1})$. Ultimately we want this sequence to reach R, which is the case once $R_j = R_{j-1} = R$. Many questions appear immediately, and we address them next:

- what should be the initial tentative result R_0?
- how does this generalize to more complicated object programs?
- how do we know this terminates?

3.3.4. Initializing results

We first discuss how to initialize the result of our fixed point computation. Let's not lose sight of what we're trying to do: computing the set of possible results from a call. Note that we are speaking of a set of results, for simplicity, even though technically results are input-output relations. A relation is also a set (of pairs), and we ignore the distinction here.

The desired set is subject to a tension between two goals: correctness and accuracy. A set is correct if it includes all the values that may occur there during ordinary execution. In this sense, a full set (completely unconstrained) - is always trivially correct. In the opposite direction, a smaller set is more accurate: it carries more (partial) information about what the results can be. Therefore, we are after the smallest correct set. Conceptually this set certainly exists, because it contains the results that are actually possible during ordinary execution. The only question is how close we can get to that conceptual set.

Now observe that when traversing a *pgraph* node, partial interpretation proves that certain results are possible, based on the inputs that are possible. In other words, intuitively the logic of the object program *adds* possible values to sets. It therefore makes sense to start tentatively from a small set, and let it grow until we reach a fixed point.

We use the empty set as the initial tentative result from a call. We interpret the empty set as "the result of" non-termination. We justify this as follows. The result set indicates the possible values that the call may return (during ordinary execution). If the set is empty, then the call may not return any value at all. This is only possible if the call does not return.

Since the empty set can now appear as the result of a call, all *pgraph* node types need to handle the empty set as an input, as well. Continuing with the non-termination interpretation, most node types simply produce the empty set, if any of their inputs are the empty set. Only *if* deviates from this principle, because it is non-strict.

At an *if* node, if the condition is the empty set, then the net result is also the empty set. Otherwise, we attempt to follow the steps from 3.2.7. First, we attempt to decide the condition. If we succeed, we directly report the result from the correct branch. If we cannot decide the condition, our plan is to construct a conditional expression. This is not a problem if both branches (consequent and alternate) report a non-empty set. However, in the case when at least one branch reports the empty set, we must do some additional work.

If both branches (consequent and alternate) report the empty set, then the net result of the *if* node is also the empty set.

If only one branch reports the empty set, there is a complication. Recall the propositional reformulation of the conditional expression:

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

This makes sense as long as we have expressions a and b, from the two branches. However, if a branch reports the empty set, we *do not* have an expression for the possible values from that branch. Suppose for the sake of exposition that the alternate branch reports the empty set. Then whenever $\neg c$, y can take *no values at all.* We can write this symbolically as follows:

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

This algebraic manipulation shows that when the alternate branch is the empty set, 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 the empty set: 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 set of the *if* node describes the possible values coming out of the *if* node, *if the if node terminates* - as all our result sets do. If the *if* node terminates, but the alternate branch would not terminate (i.e. it reports the empty set), then we *do* know that the condition was true. This justifies the assertion of the condition in the result - intuitively, as well as algebraically.

We clarify that in this section, we are not deciding termination. When a $\perp$ result arises (i.e. the empty set), it is often transient in the early stages of the fixed point computation. If a result *does* converge to $\perp$ (e.g. `p(x) = p(x+1)` for a completely unknown `x`), then we have indeed proven *non*-termination. However, if a result converges to a non-empty set, we have *not proven termination*; we have merely shown that *if* the operation terminates, its actual result (during ordinary execution, later) is constrained to be a member of the result set. And that is exactly the point we *are* making in the previous paragraph: *because* the result set has this meaning, it is correct to assert the condition one way when a branch reports $\perp$.

3.3.5. Generalizing to more complicated object programs

Summarizing, we compute a sequence of tentative results from a call. This sequence begins with the empty set, and progressively grows larger.

In the example of `fact(x)`, we are computing a single sequence of tentative results: those from the call to `fact(x)` - this is the only call that occurs in the example.

In general, we need to compute several sequences of tentative results - one for each pair `(pgraph, args)` corresponding to a call that occurs during partial interpretation.

We store these tentative results in our cache entries, indexed by `(pgraph, args)`. However, we do not store an entire sequence of tentative results, in a given cache entry; instead, we store only the most recently computed result. Intuitively, this most recently computed result "contains" all the information from the previous results. Literally also, the most recently computed set contains all previously computed sets, in the sense of set inclusion. This fact will be highlighted (and proven informally) in our termination argument.

Again in the example of `fact(x)`, we mentioned only loosely how to perform re-computation of the result. We pointed to part of the pgraph of `fact`: the data path from the inner return, to the outer return - we referred to this data path symbolically as F. This data path depended on a single result: the previous result from `fact(x)`.

In general, re-computation of a result can involve an arbitrarily complex data path through a *pgraph*. This data path can depend on any number of results, stored across as many cache entries.

To avoid this complexity, we perform re-computation by performing partial interpretation of the entire *pgraph*. This is done as described in 3.2, with one important modification. When we come to a sub-call, we read the result currently stored in the cache for that call; we no longer recursively invoke the interpreter at sub-calls. In effect, this amounts to computing a result which is up to date with respect to all the results currently in the cache.

We complete this with an outside "driver loop", which repeatedly re-computes the results of calls, until we reach a fixed point over all the results in the cache. This marks a substantial change in the structure of partial interpretation. Previously in 3.2, we described partial interpretation as the mapping from each data node in a *pgraph*, to a computational history. This is no longer the whole of partial interpretation - if we need to make the distinction, we will refer to this previous process as *pgraph partial interpretation*. It remains a core piece of partial interpretation: it re-computes the result from a call, thereby advancing one step towards a fixed point.

We could traverse the entire cache, re-compute every result, and repeat until none of the results change. Instead, to avoid needless re-computation, we track the dependencies between cache entries - more precisely, between the results they contain.

In each cache entry e, we additionally store a list of cache entries whose analysis consumed the result of e. Whenever *pgraph* partial interpretation reads a result of a cache entry e, the current execution is registered as a consumer of e. We also introduce a global worklist, which is a list of cache entries, each of whose result needs re-computation - because there has been a change in some of the results they consumed. Whenever we complete partial interpretation of a *pgraph*, if the result has changed, we put its consumers on the worklist. Also, whenever we encounter a *call* for which there exists no cache entry, we create a fresh cache entry, initialize its result to the empty set, and put this cache entry on the worklist.

The top-level partial interpretation algorithm is then as follows. To perform partial interpretation of a root call, we first create a cache entry for this call. We initialize this cache entry's result to the empty set, and put it on the worklist. Then, we flush the worklist: until the worklist is empty, we pull a cache entry from it, and run *pgraph* partial interpretation to re-compute its result. Once the worklist is empty, we have reached a fixed point. We can now read the result from the cache entry we created for the root call.

3.4. Termination

We now have the right overall structure for our partial interpreter, but we are not done. It is time to discuss the termination of partial interpretation. Our plan now is to walk through an informal termination argument. Along the way, we encounter three parts where the argument fails, as it stands. We label these three parts P1 through P3, and address them in separate subsections (3.4.4 to 3.4.6). In these subsections, we describe the modifications we make to partial interpretation, to satisfy the requirements of the termination argument. The purpose here is first to identify where such modifications are needed, and to outline the design space for such modifications, i.e. the exact requirements of the termination argument. We do not have principled solutions for these modifications, so we then select *un*principled solutions which we use in our implementation.

This contrasts slightly with our presentation so far. We have been motivating modifications to the partial interpreter, with examples of failure to terminate. These examples (of object programs) were simple and illustrative. From this point forward, we find it is better to focus on the generality of a termination argument, rather than the specifics of examples. The examples would be more complicated, and detract from the important ideas.[7]

3.4.1. Overall argument

We begin our termination argument. We visualize an execution of partial interpretation, on a two-dimensional grid. Horizontally along the bottom, we place all the cache entries that arise during partial interpretation. Vertically over each cache entry, we place the sequence of

[7]Also, in some cases we do not know what the examples would look like.

results computed for that cache entry. Intuitively, this is delimited by a rectangle, whose area corresponds to the total number of re-computations made during partial interpretation. Our overall argument is that the area is finite, therefore partial interpretation terminates. The area is the product of the lengths of the sides (horizontal and vertical), and we argue their finiteness separately. These sub-arguments are substantial, but the overall argument should be intuitively clear: if we create finitely many cache entries (horizontally), and re-compute their results a finite number of times (vertically), then we terminate in finite time.

Before we go into the sub-arguments (horizontal and vertical), we comment on the intuition of the rectangle. For a given object program, the shape on the two-dimensional grid might not be an exact rectangle: some results might get re-computed more than others. However, consider the worst case: every result depends on every result. Therefore any time one result changes, all the results must be re-computed. Continue to consider the worst case: when we re-compute all the results, only one result changes. In the absolute worst case, we re-compute this result last, such that all the others were re-computed "for nothing".

This worst case behavior justifies the intuition of the rectangle. On each row, we re-compute all the results, and a single one changes. If no result changes, we get two successive rows which are identical, and we have reached a fixed point. We can fit any more optimistic execution within this rectangle. If there are less dependencies between the results (than in the worst case), we can conceptually leave some entries blank in a row - or copy them from the row below, without actual re-computation. If more than one result changes on a row, we need less rows to reach a fixed point.

We first argue vertical finiteness. We start by reducing vertical finiteness to something simpler. If we know that each *individual* result can only change a finite number of times, then the total number of rows must be finite. This is because one result changes on each row. So the result of the first cache entry can only change so many times, then the result of the second, and eventually we run out - assuming horizontal finiteness also, which we will argue in 3.4.6. Specifically, suppose we have m cache entries, and the result of each can change at most n times. Then the number of rows is at most mn: each result changes n times. The total number of re-computations is at most m^2n: m re-computations take place on each row.

We argue that a given result can only change a finite number of times, in three parts:

- a *directionality* in the changes: when a result changes, the new result strictly contains the old, in the sense of set inclusion - in technical terms, the result is monotonically increasing under set inclusion
- the existence of an *endpoint* in that direction: the full set
- finitely many steps before we reach this endpoint.

Before we argue directionality and finitely many steps, we comment on the endpoint. The endpoint exists trivially (the full set), but it is significant. Once we've reached the full set for a given result, directionality ensures that the result can no longer change. In other words, this particular result has reached its fixed point.

3.4.2. Result directionality

Here we are explicit that a result is an input-output relation. We show that when we (re-)compute a result, it either remains the same or grows larger. We first introduce a strong induction on the global (chronological) order of (re-)computations. In other words, to prove that result directionality is respected by a re-computation, we assume that it has been respected by all *previous* (re-)computations, of any cache entries. We will use this assumption in one specific place, with the phrasing "we appeal to our strong induction hypothesis".

In the case of the first computation for a cache entry, directionality is trivial because the previous result was empty. In the case of a re-computation, we examine the *pgraph* whose result is being re-computed. We show that directionality is respected *at every node*. Once we have shown this, we observe that in particular it is respected at the *exit* node.

The formalism we use here may not be the clearest, but our goal is to identify where the argument fails, and the corresponding modifications to our algorithms; not the formality of the argument. We refer the reader to our article [**SMF21**] for a clearer formalism and proof, based on a relational semantics. The relations there are more like the ones found in databases and formalized in [**BCD14**]. Each element of a relation is a possible concrete execution. Composition becomes relational join. The same proof steps shown here are used in the new formalism.

Lemma 3.4.1. *(Node directionality) Suppose that in a (re-)computation for a cache entry, a pgraph node n got mapped to a result R (a relation from the formals to n). Then in a subsequent re-computation for the same cache entry, n gets mapped to R', with R ⊆ R'.*

PROOF. (SKETCH) For a few node types, $R = R'$ is fixed: at *entry*, R is the identity relation; at *const* and *prog*, R is a constant relation. For the other node types, we proceed by structural induction on the *pgraph*. We suppose a previous relation S from the formals to the successors of n, and a current relation S', with $S \subseteq S'$.

We now write R as a composition of relations $S; L$ (S followed by L), where L is a local input-output relation at n. For any relations S and L, we state without proof that:

$$S \subseteq S' \wedge L \subseteq L' \;\rightarrow\; S; L \subseteq S'; L'.$$

We already know $S \subseteq S'$ (by structural induction), so it suffices to prove $L \subseteq L'$.

For many node types, the local relation $L = L'$ is fixed:

- *prim*: for example, the relation for $+$ is $\{(a, b, a+b) \mid \text{for any } a, b\}$
- *if*: $\{(\texttt{true}, a, b, a) \mid \text{for any } a, b\} \cup \{(\texttt{false}, a, b, b) \mid \text{for any } a, b\}$
- *exit* and *result*: all these do is pack and unpack data component-wise. Their local relations are therefore fixed. For example, the relation for an *exit* of three inputs is $\{(a, b, c, (a, b, c)) \mid \text{for any } a, b, c\}$.

Only *call* remains, and here L can differ from L', since these are results which we read from the cache. Conceptually, we first compute the argument set A, to see which cache entry we read from. A is the image of the set of formals, under S. There are two main cases. First, if $A = A'$ (the arguments have not changed), we read L' from the same cache entry e as before. Either e has been re-computed since we read L, or it hasn't:

- if e has not been re-computed, $L = L'$
- if e has been re-computed, we appeal to our strong induction hypothesis: $L \subseteq L'$.

If $A \neq A'$, we encounter a problem. We are reading a result from a *new* cache entry e'. Even though we know $A \subset A'$ (since $S \subseteq S'$), we have no guarantee that the L' we read from e' includes the L we read from e. In the extreme case, e' might be freshly created, in which case its result is the empty relation. If this happens, node directionality is broken, result directionality can be broken, and so can termination. We label this problem P1, and address it in 3.4.4. Assuming suitable handling of P1, *call* respects directionality. $\square$

3.4.3. Finitely many steps in the result

We want to argue that a result can only grow a finite number of times, before reaching the full set. As it stands, this is not true. We examine an example briefly. Consider once again the partial interpretation of `fact(x)` - factorial of a completely unknown integer `x`. Initially the result is the empty set. During the first computation, in the alternate branch we read the empty set at the sub-call. We therefore get the following result, schematically:

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

$\perp$ stands for the empty set, and we allow ourselves to write $\perp$ in a branch of a conditional expression. The next result is:

$$\begin{array}{l} \text{if } x \leq 1 \text{ then } 1 \text{ else} \\ \quad \text{if } x = 2 \text{ then } 2 \text{ else } \perp \end{array}$$

Then:

$$\begin{array}{l} \text{if } x \leq 1 \text{ then } 1 \text{ else} \\ \quad \text{if } x = 2 \text{ then } 2 \text{ else} \\ \quad\quad \text{if } x = 3 \text{ then } 6 \text{ else } \perp \end{array}$$

etc. We are progressively unrolling the execution of factorial into conditional expressions, and this never reaches the full set. We label this problem P2, and address it in 3.4.5. With suitable handling of P2, there are finitely many steps from any result to the full set.

Summarizing, in the big picture of our termination argument, we have just argued vertical finiteness: results are re-computed (or, they grow) finitely many times. The next piece is horizontal finiteness: finitely many cache entries are created. We label the entire question of horizontal finiteness P3, and address it in 3.4.6. Combining horizontal and vertical finiteness, we have argued that partial interpretation terminates. Next, we address the three missing pieces of the argument, which we have labelled P1 through P3.

3.4.4. Handling reading from a new cache entry [P1]

Recall that this problem arises in the argument for directionality of the *call* node - within the argument for result directionality. Specifically, the problem arises when the arguments to the *call* node have changed. The problem is that we read from a different cache entry than before. As it stands, we have no guarantee that the result there is larger than the last result we read (from the old cache entry).

We take some time to discuss this problem, but we do not know the final conclusion. Although there are various ways to ensure that this problem does not break termination, we do not know all their implications, especially on *what we are computing.* We first offer some insight into the problem. We briefly dismiss three approaches as unsuitable. We then discuss two approaches to addressing the problem. Finally, we explain why we ignore the problem in our implementation.

In some sense, the problem is that we are attempting to re-use a standard template for an algorithm and a termination argument, even though our setting does not quite fit. In a traditional fixed point computation, a fixed re-computation function is applied repeatedly to some value, and it is the fixed point of this function that we are computing. In our setting, the "value" is the tuple of all the results stored in the cache, and this is fine. Each component in the tuple depends on any number of components, which corresponds to a re-computation equation for each component. This implies a re-computation function for the whole cache, *given the dependency relations between the components.*

What does not quite fit is that in our setting, this re-computation function *is not fixed.* As the components of the tuple grow, so can the arguments to sub-calls, which can *change the dependencies between the cache entries.* However, there is some structure to this evolution of the dependencies: always towards cache entries whose arguments are larger.

If the arguments are larger, intuitively the results must be larger. We can formulate this as a kind of *congruence* over cache entries. First notice that there is a partial order on cache entries (whose *pgraph*s are equal), under argument inclusion. Then, given two cache entries (with arguments) A and B, and their results R_A and R_B, the congruence is: $A \subseteq B \rightarrow R_A \subseteq R_B$. Notice that this congruence generalizes our caching principle, which was: $A = B \rightarrow R_A = R_B$.

So far, our algorithm and termination argument have been framed as though the re-computation equations are the only constraints over the final results, and the only means of advancing towards those results. However, we see that this is not enough to complete the termination argument. In some sense and to some degree, we must also include *congruence* of results as a constraint over the final results, and as a means of advancing towards those results. This idea is at the heart of the two approaches we propose to handle the problem.

We dismiss the idea of ordering re-computations so that the problem does not arise, since this idea appears to fail for suitably circular object programs. We dismiss the idea of intervening to enforce result directionality at the end of a re-computation, since it would no longer be clear what we are computing. We dismiss the idea of intervening to enforce node directionality based on results previously read at that node, since two executions could report a different result when reading from the same cache entry.

A first approach is to *propagate selected congruence information*, between cache entries. Suppose (the analysis for cache entry) C previously read from (cache entry) A, and now reads from B, with $R_A \nsubseteq R_B$. We can intervene, compute the union $R_A \cup R_B$, *store it in* B, and report this as the result of the *call*. Roughly speaking, we are using R_A to "initialize" R_B. In other words, we are accepting that re-computation is not the only means by which cache results change; we are accepting result congruence as another such means. Specifically, we are propagating exactly enough congruence information to ensure node directionality. Note that result directionality is respected when we write the union into the cache. Therefore we maintain the validity of our overall termination argument.

A second approach takes our first approach to its logical conclusion. In the first approach, we have already accepted that results can change by virtue of congruence; not just by virtue of re-computation. At this point, it seems plausible to *propagate all congruence information* between cache entries. We can achieve this by making two modifications to our algorithms. First, when a cache entry E is created, we can find every cache entry whose arguments *are included in* E, and initialize R_E to the union of these results. Second, when we finish a re-computation for a cache entry E, we can find every cache entry E' whose arguments *include* E, and set R'_E to $R_E \cup R'_E$.

This second approach seems natural. Further, it can be seen not just as addressing an undesirable phenomenon at a *call* node. Looking at it differently, it can be seen as taking advantage of an *opportunity*: we are leveraging congruence to speed our way to an eventual fixed point. There is no question that computing a union is faster than re-executing an entire *pgraph*. However, here again it is not clear to us *what final result we are computing.*

In particular, we do not know if the same final result is achieved by propagating *selected* congruence information (first approach), or *all* of it (second approach).

These questions require further investigation, but we do not conduct it in this work. In practice, we have not encountered a program where all this is a real *termination* problem. [**Ruf93**]'s section 4.4.2 describes features of programs where arguments grow at a call, breaking (what we call) node directionality. It is not clear to us what is additionally needed to break *result* directionality; and even less clear, what is needed to break termination. For these reasons, our implementation currently ignores this problem.

3.4.5. Finitely many steps in the result, revisited [P2]

Recall that this problem arises in the argument for vertical finiteness: it is complementary to result directionality. The problem is that although results can be changing directionally towards an endpoint (the full set), they can change infinitely many times without reaching the full set. We briefly examined the example of `fact(x)`, in 3.4.3.

A first observation is that our language of set comprehension does not admit a finite expression of the exact result from `fact(x)`. Therefore the problem is not merely one of "converging to the answer too slowly"; there is no answer to converge to. This is not the end of the world. For the purposes of termination, we can tolerate convergence to an approximate result: a set larger than the exact result - even the full set. Therefore this observation is not the crux of the problem.

The key feature of the problem is that we can grow a set indefinitely, without reaching the full set. In technical terms, in the *lattice* of all the sets expressible with our comprehensions, there are *infinite ascending chains*[8]: infinite sequences of progressively larger sets, which never reach the full set - such as the sequence of results from `fact(x)`.

In our assessment, the choice of solution to this problem is a critical aspect of partial interpretation. We have not explored the possibilities very much, and it's not even clear to us what the range of possibilities is. Nevertheless, we discuss three points. First, we describe some general features that any solution must exhibit. Second, we mention one class of solutions which exhibit these features. Third, we present a simple solution which we use in our implementation.

Any solution must eliminate infinite ascending chains. This is enough to fit within our termination argument. This elimination can only be achieved by performing some sort of adjustment of a result, after that result has grown. If there is no adjustment, we'll get the original (infinite) sequence. To maintain correctness, any adjustment must be towards a larger set. In other words, it is necessary to make a conservative approximation of the result - information must be discarded.

[8]This terminology refers to the common visualization of a lattice as a Hasse diagram, with the empty set $\perp$ at the bottom, and the full set $\top$ at the top.

One disciplined way to discard information is to introduce a new lattice, which we refer to as an *approximation lattice*. This lattice contains sets which approximate the sets in our original lattice. Additionally, the approximation lattice is constructed to have no infinite ascending chains. We can then modify our algorithms: any time a result grows, we can "round it up" to a set in the approximation lattice. In technical terms, this "rounding" corresponds to a notion of *widening*, from *abstract interpretation* - we adopt this terminology.

There are many possible approximation lattices. The choice of approximation lattice determines which information is kept and lost, upon widening. In principle, we could choose a different approximation lattice at each call. We could then tailor these choices to adapt to the information needs of the caller. In particular, conceptually we could preserve any finite combination of boolean conditions.

In this work, we do not explore this idea of "adaptive widening". Instead, we present a simple mechanism for widening (i.e. adjusting) results. We stress that this particular choice is not very important, in the big picture of partial interpretation. It mostly illustrates that it is possible to instantiate the description above, and it allows us to experiment with partial interpretation in practice.

We widen results as follows. First, we do not widen at all if the previous result was the empty set. This is a practical choice which leaves unwidened the result of any first computation. This accounts for many results, in practice. When we do widen, we begin by discarding all relational information between formals and results. We do this by explicitly quantifying over the formals, which turns the result into a set. Then, we widen component-wise: one result at a time (among possibly multiple results), and one item at a time in data structures. This effectively discards all relational information between components.

We widen a component based on the following three ideas:

- preserve information about uniquely known "scalars" (booleans, integers, programs)
- discard all information about non-uniquely known scalars
- preserve information about the known shape of data structures - in our case, of lists.

We guide this widening with the known types of set elements. To widen a scalar, we test it for uniqueness. We then leave it unwidened if it is uniquely known; otherwise, we replace it with a completely unknown scalar. To widen a pair, we recursively widen its two components. To widen a list, we first check if the empty list is a possible value. If it is, we further check if it is the only possible value. If so, we leave it unwidened. If the list can be both empty or not, we replace it with a completely unknown list. If the list cannot be empty, intuitively we have some information about the shape of the list, and we preserve that information. We do so by recursively widening the head and tail of the list.

Under this widening of results, a result cannot grow forever without reaching the full set. We first give an intuition for this argument; then, we make it more precise as a structural induction on types.

Intuitively, notice that any widened result can be seen as a data structure of known shape, containing some number of "slots" for scalars. A scalar is in some sense a trivial data structure, with a single slot. There are two ways for the result to grow (from a widened result to another widened result). First, we can pick a scalar slot which was uniquely known, and make it completely unknown. Since there are finitely many slots in a data structure of known shape, we can only do this finitely many times. Second, we can throw out some information about the shape. In our programs, only lists can have unknown shape, given their type. When we lose information about the shape of a list, this happens at the tail. The known segment shrinks to a shorter list, and we can only do this finitely many times.

More precisely, we show that a given value can only grow a finite number of times, by structural induction on the type of this value. A scalar (boolean, integer, or program) can grow 0 or 1 times - if it is unknown or known, respectively. In a pair, each of the two components can grow finitely many times (by induction), and we sum the two numbers. In a list, there are three cases. First, if the list is known to be empty, it can grow a single time - to the completely unknown list. Second, if the list is completely unknown, it cannot grow at all. Third, if the list is (locally) known to be a `cons` cell, then each of its head and tail can grow finitely many times (by induction), and we sum the two numbers. This induction on the tail is well-founded, because we cannot have a list of known, infinite length. This would require an infinitely large set comprehension, or recursion in our language of propositions.

Widening results can introduce new variables which express the components which have been made "completely unknown". In other words, now results may not depend exclusively on inputs. When a caller consumes such a result, it must keep those variables distinct from other variables in the caller's scope. Practically, we do this by renaming those variables to unique names in the caller's scope.

3.4.6. Finitely many cache entries [P3]

Recall that this problem corresponds to the second half of our termination argument: horizontal finiteness. We must ensure that finitely many cache entries are created.

Cache entries are (only) created when a call occurs, if there is no cache hit for that call. We must therefore ensure that finitely many pairs `(pgraph, args)` occur, at *call* nodes.

Calls - and their corresponding pairs `(pgraph, args)` - do not occur out of thin air. The key idea here is to consider the static call graph of the object program. Each call exists at an endpoint of a path through the static call graph. Each such path begins with the root call. Then either the path stops, or it continues into a sub-call from the root call, and so on.

If the object program does not contain any cycles, then all these paths must be finite (in length), because the object program is finite. However, if there are cycles (i.e. the object program contains recursion, or mutual recursion), then a path can loop around a cycle indefinitely, creating cache entries along the way.

This suggests the following two-fold plan, to ensure that finitely many cache entries are created.

First, we ensure that any sufficiently long path will always produce a cache hit. To ensure this, we make additional modifications to our data and algorithms. This ensures that there can only be finitely many paths which do not "reach" a cache hit: those that are not sufficiently long. Cache entries are only created at an endpoint of these paths. We therefore ensure that there are finitely many *execution points* where cache entries are created. Intuitively, we bound the unrolling of the call graph during partial interpretation.

Second, in general, several cache entries can be created at each execution point (in the unrolled call graph). This is because arguments can grow over the course of partial interpretation, as we have seen when arguing the directionality of *call* (in 3.4.2). We additionally ensure that arguments can grow finitely many times, by applying an argument widening when they grow. We therefore ensure that finitely many cache entries are created.

We present a general framework for ensuring that sufficiently long paths produce a cache hit. Simultaneously, we describe a concrete instance of this framework, which we use in this work and which is due to [**WCRS91**]. Here again, we stress that this particular concrete choice is not very important, in the big picture of partial interpretation.

Our general framework is in three pieces: a termination goal, a mechanism for termination assessment, and a mechanism for termination resolution. Selecting a termination goal is done once and for all, when implementing partial interpretation. Then, the two mechanisms - termination assessment and termination resolution - are applied each time a call occurs, if there is no cache hit.

First, we select a *termination goal* for partial interpretation. Ideally, we would like partial interpretation to terminate on all object programs and all inputs. In this work, we relax this goal: we aim for termination *when the object program would terminate, at run-time.* In other words, we tolerate non-termination of the partial interpreter - if the object program might not terminate during ordinary execution.

We are about to discuss termination assessment, so we make a clarification about the decidability of halting. The theoretical limits on deciding halting do not quite reflect practical reality. There are many ways to mechanically assess termination, usually falling back on reporting "don't know" in some cases. See size-change graphs [**BAL07**], especially as used in [**Ngu19**]; linear algebra methods [**Tiw04**]; and the work from Microsoft Research on the Dafny system for verified programming [**Lei10**]. For some terminating programs, Dafny infers termination metrics (decreasing quantities in a well-founded domain), expanding on classic (manual) termination proof methods going back at least as far as [**Flo67**]. Besides, in our setting the object program cannot invoke the termination checker, which voids the traditional argument for the undecidability of halting. In any case, our goal is to decide termination *as often as possible.*

Second, we select a mechanism for *termination assessment.* When a call site is reached during partial interpretation (and there is no cache hit), we decide if we need to intervene to achieve our termination goal. In this work, we use a simple criterion based on two ideas.

The first idea is that there is no risk of non-termination until we have actually looped in the static call graph. This is true regardless of our termination goal. We therefore keep track of a *call stack* during partial interpretation: a stack of `(pgraph, args)` pairs which document the calls we have made on the way to the current execution point. The stack is stored when creating a cache entry. It is restored when we begin *pgraph* partial interpretation for that cache entry. At *call*, we push the details of the *call* on the stack. In effect, the stack records the path we have taken through the call graph. This lets us identify when we have looped in the call graph: when the stack contains two calls to the same *pgraph*.

The second idea is that because of our specific termination goal, we can further refine the conditions under which we judge that there is a risk of non-termination. Not only must we have called the same *pgraph* twice, but additionally we must have crossed an undecided branch in between. If all branches have been decided, then we are executing a single code path: the same that would be executed at run-time. We will therefore terminate, *if we would terminate at run-time.* To identify this situation, we keep track of each undecided branch we encounter, by pushing an `if` marker on the call stack.

This (non-)termination criterion is referred to as *guarded recursion*: the recursion is guarded by an undecided branch. It allows unrolling all loops which are "controlled" by known data. This is critical for partial evaluation, which is why we are willing to tolerate non-termination of partial interpretation, when ordinary execution might not terminate. Tolerating non-termination is not the only option, and we discuss this again in Chapter 5.

Third, we select a mechanism for *termination resolution.* When we assess that our termination goal will not (or might not) be achieved, we intervene in some way. In this work, we intervene by adjusting the arguments to the current call. Before we describe this adjustment, we outline our plan for the argument.

We plan to argue that a path can only return to a given point p in the static call graph (crossing an undecided branch along that path), finitely many times before we get a cache hit. To argue this, we consider the sequence of arguments at each successive visit to p along a path. We then re-apply an argument template we used for vertical finiteness (in 3.4.2). We argue that the sequence of arguments exhibits directionality, that this directionality has an endpoint, and that there are finitely many steps to that endpoint.

We ensure directionality as follows. We know the stack contains a previous call to the same *pgraph* being called now. We adjust the current arguments to be the union of the previous arguments and the current arguments. This union grows the arguments, because the previous and current arguments are not equal - if they were, we would have had a cache hit without any intervention.

The existence of the endpoint is once again trivial: the full set. Eventually we must reach it, and the next argument in the sequence must be the full set again. We then have a cache hit, and we are done.

We ensure that arguments can only grow finitely many times, on the way to the full set, as follows. After taking the union (to ensure directionality), we widen. We do this similarly to how we widen results (in 3.4.5). Widening is simpler here, since we do not need to eliminate a relation between formals and results - the arguments are already a set.

This union and widening together ensure that paths have finite length, and that the unrolled call graph is bounded.

It remains to ensure that at each call site (at the endpoint of each of these paths whose number we just bounded), finitely many cache entries are created. A cache entry can be created each time the arguments grow. Once again, we ensure this happens finitely many times, by widening arguments when they grow. As in result widening, we can skip widening when growing for the first time - from the empty set to a non-empty set.

To identify when arguments have grown, we store additional information in our cache entries. In each cache entry, at each call site, we track which cache entry was read there previously. We initialize these “prior cache entries” to a null value. Thus in the first computation for the result of a cache entry, we know that no widening is needed. In subsequent re-computations, we can compare the new arguments to the arguments in the old cache entry, and widen if the new arguments exceed the old.

This ensures finitely many growths in the arguments. Note that these “prior cache entries” also lay a foundation for our “first approach” to resolving P1 (see 3.4.2 and 3.4.4): propagating selected congruence information, from the old cache entry to the new. We will also use them during specialization (in 4.1).

This completes our argument for horizontal finiteness: finitely many cache entries are created. This in turn fills the last blank from our overall termination argument, from 3.4.1.

3.4.7. Consolidated algorithm

Appendix A shows a consolidated view of our partial interpretation algorithm. There, we show which sections discuss each part of the algorithm. We also show which parts of the algorithm were introduced to satisfy our termination argument.

Chapter 4

Application to partial evaluation

We emerge from Chapter 3 with a theoretical view of partial interpretation. The details are complicated, but the overall view should not be: to perform partial interpretation of a program, we expand it into a network of cache entries whose results depend on one another; then, we find a fixed point over all these results.

Our initial implementation of partial interpretation directly reflects this theoretical view. In this chapter, we follow through with our original plan: to implement a specializer which consumes the results of partial interpretation. When we apply this specializer to larger examples, we will see that our primary concern will shift to the performance (execution time) of our algorithms, and away from the theoretical concerns of Chapter 3. Nevertheless, the theoretical view - and in particular the termination argument - will frame any ideas we have for modifying our algorithms to improve their performance.

4.1. Specialization from partial interpretation

The specializer needs to check the uniqueness of many values. We first explain how the specializer interacts with partial interpretation, to make these uniqueness checks. Then, we explain how the specializer uses this information to specialize the original program.

4.1.1. Performing uniqueness checks

Before specialization begins, partial interpretation has been completed: we have reached a fixed point over all the results in the cache. Each cache entry contains the results returned from a call, corresponding to a pair `(pgraph, args)`. This is a good start, but it's not all the specializer needs. To specialize a given call `(pgraph, args)`, the specializer additionally needs information about all the *intermediate* results computed therein.

To get this information, we run *pgraph* partial interpretation one more time - as described in Chapter 3. This time there are no complications with termination, as there were in Chapter 3. Each call site (out of each cache entry) already stores a connection to the correct cache entry (our "prior cache entries" from 3.4.6), at the partial interpretation fixed point. We handle a *call* by following that connection, and reading the result stored there.

We therefore introduce a second mode of operation for *pgraph* partial interpretation, which we refer to as *passive mode*. In passive mode, because the handling of *call* is trivial, we omit the following aspects of partial interpretation:

- we do not perform cache lookup (we have the connection to the correct cache entry)
- we do not perform adjustment of arguments: termination assessment, termination resolution, and argument widening
- we do not place `if` markers on the call stack, to check for guarded recursion
- in fact, we do not keep track of a call stack at all
- we do not write results into cache entries.

As in 3.2, *pgraph* partial interpretation (in passive mode) produces two items: a mapping from nodes to extended comprehensions, and an SMT process which is incrementally updated with the declarations and propositions from those extended comprehensions. The specializer accesses these two items directly. It can then check for uniqueness at a given node, as follows:

- get the extended comprehension associated with this node
- extract the expression from the extended comprehension
- query the SMT process for the uniqueness of that expression.

4.1.2. Using uniqueness information

We approach the specialization of a *pgraph* as the construction of a new *pgraph*, whose nodes are written as we traverse the original *pgraph*. We refer to this general construction process as *transcription*.[1] Transcription is an instance of our general maneuver for defining algorithms over *pgraph*s (see 2.2): a node n in the original *pgraph* maps (possibly) to a node $f(n)$ in the new *pgraph*. It remains to specify the local handling for each node. Then, we apply f to the *exit* node, and we get a transcribed *pgraph*.

We find it useful to first consider the simplest possible local handling: write the same node label, and edges to the (transcribed) images of our successor nodes. This performs no specialization; it constructs an isomorphic copy of the original *pgraph* - with possibly re-arranged node indices. Next we describe how specialization deviates from this, to avoid transcribing some nodes from the original *pgraph*. The simple "copy" behavior remains relevant, even during specialization: sometimes we can do no better than to reproduce a node from the original *pgraph*.

The specializer uses uniqueness information in three ways, to do better than just reproduce the original *pgraph*.

[1]We also like "rewriting", but the technical phrasing "graph rewriting" already refers to making local transformations to part of a graph.

- If a *result* node has a unique value, then we replace it with a *const* (or *prog*) node containing that unique value. As an immediate consequence, the producing operation for this *result* is not needed, so we do not request its transcription. Therefore we also avoid the transcription of its (transitive) dependencies. Because transcription starts at the *exit* node, we are thus avoiding the transcription of an entire subgraph.

If a *result* is not unique, we apply the "copy" behavior. Even when a result is not unique, we can make two further improvements.

- We extend the idea of avoiding the transcription of a subgraph, to the *if* node. If the condition is uniquely known, we transcribe only one of the two branches (consequent or alternate). If the condition is unknown, we apply the "copy" behavior.
- If we have a *call* to a unique *pgraph*, we can specialize that *pgraph*, and write a call to the specialized program. There is a complication here: if we blindly "specialize recursively", specialization might not terminate. For example, specialization of `fact(x)` - factorial of an unknown integer - would not terminate. We resolve this shortly.

Other node types apply the "copy" behavior.

To resolve the termination of specialization, we cache specializations: we map a pair `(pgraph, args)` to a specialized program. We achieve this by adding a field to our cache entries. This field holds a reference to the (eventually) specialized program, for that pair `(pgraph, args)`. This works because we only ever specialize `(pgraph, args)` pairs for which we already have cache entries from partial interpretation.

4.1.3. Discussion

We have just described a direct interaction between specialization and *pgraph* partial interpretation: specialization runs *pgraph* partial interpretation (in passive mode), to check the uniqueness of values. Both of these algorithms are *pgraph* traversals, and we discuss how they relate.

Specialization begins "from the end" (at the *exit* node), and almost immediately it checks if *result* nodes have a unique value: those which *exit* depends on. If some of these have unique values, the specializer will avoid transcribing the subgraph(s) rooted there. However, these subgraphs *are examined*: not by the specializer, but by *pgraph* partial interpretation. The specializer checks uniqueness by invoking the memoized *pgraph* partial interpretation function. To produce a result, *pgraph* partial interpretation must traverse the entire subgraph, memoizing all the results on the way. Consequently, no cost will be incurred when the specializer *subsequently* queries *pgraph* partial interpretation.

This highlights the distinct nature of specialization and partial interpretation. Specialization fundamentally works backwards from the end of a program, attempting to avoid as much transcription as possible. In other words, specialization is a memoized depth-first traversal which may be truncated depending on the results of partial interpretation. Partial interpretation fundamentally works forward from the beginning of the program, propagating partial information through the logic of the program. To make a decision near the end of the program, specialization needs the result of partial interpretation, so it "pulls" partial interpretation all the way there.

This distinction is at the heart of the accuracy gains we get over previous partial evaluators. Our specializer cares only about the uniqueness of a given result; not about the uniqueness of *any intermediate results used to compute it.* This enables a phenomenon we call *uniqueness at a distance.* We will illustrate this phenomenon in 4.2.4.

Specialization and partial interpretation are further distinguished by the part of the cache they interact with. Although both specializations and returned results are stored in the same cache, and indexed by pairs `(pgraph, args)`, some cache entries might not get specialized. We have mentioned that when partial interpretation is complete, each call site (out of each cache entry) is connected to the correct cache entry. However, there may additionally be disconnected entries in the cache, which were relevant on the way to the fixed point, but are no longer relevant. Such disconnected entries are never visited by specialization. The only relevant entries are those connected from the root call, at the fixed point.

Of these relevant entries, only some will get specialized: sub-calls whose result is *not* uniquely known. When the result of a call is uniquely known, the specialization rule for the *result* node (given earlier) implies that we do not write a *call* at all. It is only when the result of the call is not uniquely known that we write a *call*, in which case we specialize the *pgraph* to be called.

Once we have computed all the necessary specializations, we perform a straightforward inlining to consolidate them. Our inlining is another instance of transcription, whose details we omit. We can then observe a result, in the textual format from 2.4.

4.2. Examples

We apply our system to three kinds of examples.

- First, we go over three classic, simple examples of partial evaluation. These confirm that we reproduce basic results.
- Then, we show a small synthetic example which illustrates the accuracy we gain, over previous partial evaluators.
- Last, we examine another classic, but larger example of partial evaluation. This shows that our system produces good specialized programs. However, it also shows that our system is slow.

We then begin to address the performance of our system, in 4.3.

4.2.1. Integer power

Integer power is the problem from our introduction (1.1): computing x^n by recursively examining the parity of n. We express this computation as a program:

```
(define (power x n)
  (let* (((q r) (div n 2))
         (sqr-root (power x q)))
    (if (= n 0)
        1
        (if (= r 0)
            (* sqr-root sqr-root)
            (* x (power x (- n 1)))))))
```

And here is the output of our system, for $n = 8$:

```
(define (p0 x0 x1)
  (let* ((v0 (* x0 1))
         (v1 (* v0 v0))
         (v2 (* v1 v1)))
    (* v2 v2)))
```

Two things stand out. First, all the explicit logic involving n has disappeared; it is now "baked into" the program, which squares the first input three times, without looking at the second input. This is possible because the *choices* in the original program depended only on the second argument, which was known. These choices have all been made once and for all, and what remains (the *residual program*) contains only multiplications. This is typical of the results of partial evaluation.

Second, an unnecessary multiplication by one is performed. This illustrates that "checking for uniqueness" does not account for all of program optimization: `x0 * 1` does not have a *unique value*, so our system does not simplify it. We could add some special handling for multiplying by one, then for adding zero, etc. However, this would deviate from our original strategy of grounding our methods in modern automatic deduction. Instead, we hope to complement our methods with principled program optimization, in future work.

4.2.2. Lookup in an association list

Here we lookup x in a list of pairs $((l_0 \,.\, r_0), (l_1 \,.\, r_1), ...)$. We return the first sub-list whose head matches x on the left. If there is no match, we return the empty list. This definition avoids the need for programmer-defined types, which *pgraph*s lack.

```
(define (assoc x lst)
  (if (= lst nil)
      nil
      (if (= x (fst (head lst)))
          lst
          (assoc x (tail lst)))))
```

We suppose that x is known, as well as all the left-hand sides in the list; but none of the right-hand sides. This example is significant, because some earlier partial evaluators have experienced difficulties with such "partially known structures".

Here is the output of our system, for $x = 2$ and the list $((0 \ . \ r_0), (1 \ . \ r_1), (2 \ . \ r_2))$. The right-hand sides are completely unknown.

```
(define (p0 x0 x1)
  (tail (tail x1)))
```

The specialized program goes straight for the correct sub-list. The first argument is no longer referenced. No checks are made on the shape of the list, or on its left-hand sides.

4.2.3. Map over a list

Here we map a squaring function over a list of integers. This illustrates the specialization of higher-order programs.

```
(define (map f lst)
  (if (= lst nil)
      nil
      (cons (f (head lst)) (map f (tail lst)))))

(define (sqr x)
  (* x x))
```

Here is the output of our system when specializing `map`, given that `f` is known to be `sqr`, and the list is completely unknown:

```
(define (p0 x0 x1)
  (let* ((v0 (head x1))
         (t0 (cons (* v0 v0) (p0 p1 (tail x1)))))
    (if (= x1 nil) nil t0)))

(define (p1 x0)
  (* x0 x0))
```

The squaring function has effectively been inlined into the recursion. Note that `p0` never references its first argument `x0` (the higher-order argument, formerly `f`); in the recursive call, the known program `p1` (formerly `sqr`) is passed - but it is never used.

None of the previous examples are novel results, but we confirm that we correctly reproduce them. These examples are processed in negligible time.

4.2.4. Uniqueness at a distance

We now examine a synthetic example which illustrates the increased accuracy of our methods. Previous partial evaluators focused on the uniqueness of values, *locally*. For example, if the inputs to `+` are unique, then their sum is unique - but their sum was considered unique only then. This is sufficient accuracy in some examples, but not all.

Instead, our system propagates the partial information about values, "far" across the program. This allows our system to find that values are unique *remotely*, as we illustrate with the following example.

It is a mathematical identity that $(x+1)^2 = x^2+2x+1$. Therefore, even for a completely unknown integer x, the expression $(x+1)^2 - x^2 - 2x$ has the unique value 1; despite its sub-expressions not having a unique value. Our system determines this. Here is that computation expressed as a program:

```
(define (mathident x)
  (let ((a (+ x 1)))
    (- (- (* a a) (* x x)) (* 2 x))))
```

And here is the output of our system, when specializing that program for completely unknown x:

```
(define (p0 x0) 1)
```

The reader may object that this example is not convincing, because the programmer knows this, and would not write the program like this. This is a fair objection. However, to begin with, the programmer might not notice the opportunity for simplification.

More importantly, in principle the same phenomenon can occur across a call boundary: imagine that one sub-program (say, A) computes $(x+1)^2$, x^2, and $2x$, and passes these as three arguments to another sub-program (say, B). Then imagine that B performs the same subtraction as above. We believe that conceptually, such a situation is a credible opportunity for specialization.

Unfortunately, for reasons that will be made clear in 4.2.7, our system cannot perform this "cross-call" specialization; we would need an improved notion of widening (discarding information) at call boundaries. However, we do perform this specialization "intra-call", and we can talk about the distinction.

4.2.5. First Futamura projection

We now turn our attention to a much larger example, to further validate our system's results and to evaluate its performance - the execution time of partial interpretation and specialization. We select another classic example from the partial evaluation literature: the "Futamura projections"[2] are the historical name given to a family of interactions between a specializer and an interpreter.

To re-iterate, our main objective is to stress test our system; for us, performing a Futamura projection is a means to that end. However, the maneuver is interesting in its own right, so we take a moment to discuss it. In particular, the maneuver illustrates the extent to which specialization can transform a program.

The setting for a Futamura projection begins with two program representations (i.e. programming languages), which we refer to as P and Q. We then suppose the existence of:

- an interpreter for Q, written in P;
- a specializer for P, written in any language.

Consider first the Q-interpreter (written in P), alone. Its inputs are a program in Q (a "Q-program"), as well as the inputs to that Q-program (some "Q-data", perhaps packaged as a list). The output of the Q-interpreter should be the output of the Q-program for those inputs - more Q-data. The Q-interpreter's signature therefore looks like this:

$$Q\text{-program} * Q\text{-data} \rightarrow Q\text{-data}$$

Now consider what happens if we specialize the Q-interpreter (written in P), for a known Q-program, but for unknown inputs to that Q-program. The result is an interpreter specialized for that particular Q-program. In other words, the result is a P-program which exactly mirrors the execution of the Q-program. In yet other words, the result is a translation of the Q-program, from Q to P. This is the key feature of a Futamura projection: translation by specialization of an interpreter.

To better understand how and why this works, consider the implementation of any interpreter. Any interpreter (written in P, say) fundamentally contains two parts:

- pre-existing code in P which implements each possible operation in the language to be interpreted
- further code in P which repeatedly examines the program to be interpreted, and selects which pre-existing code to execute.

If the program to be interpreted is known (the Q-program), then conceptually all these selections can be made once and for all. This is what the specializer does - just as in the integer power example from 4.2.1. What then remains of the interpreter (the residual

[2]The original reference is [**Fut71**], but our presentation is intended to be self-contained.

interpreter) is the correct code in P, to execute the Q-program. We have therefore removed exactly one layer of interpretation - we have translated from Q to P.

What we have just described is the so-called "*first* Futamura projection". It is by nature inefficient. In some sense, we are performing translation by applying *two* layers of interpretation: the interpretation of Q in P, *and* the specialization of the interpreter, whose key step is another interpretation - partial interpretation. This inefficiency is not a problem for us, because our objective is to stress test our system; not to translate between languages.

4.2.6. Our P and Q languages

We now adapt the theoretical exposition from the previous section, to our setting. We have a specializer for *pgraph*s, written in Scheme. Therefore *pgraph*s are our P language. To perform a Futamura projection, we must select a language Q, and implement an interpreter for it, written as a *pgraph*.

Because *pgraph*s are limited, we introduce a very limited language Q of arithmetic calculations. The only data in Q are integers, which are directly represented by integers in P. Programs in Q receive arguments and return results. Constant values are supported. There is no recursion, and no conditionals.

In effect, the Q language we have just described is a restriction of *pgraph*s: with only the node types *entry*, *exit*, *result*, *const*, and *prim*. We can express such a restriction of *pgraph*s as *pgraph* data: with just (statically typed) integers, pairs, and lists. We consider this a testament to the simplicity of graphs as a program representation.

Next, we define an interpreter for Q, written as a *pgraph*. The interpreter spans about 200 lines of code[3] (in the textual format from 2.4), including many comments. We attempt to remain faithful to the structure of our interpreter (from 2.3). We do not wish to adopt an "artificial" coding style, to manipulate the results of the specializer. We use a polymorphic `map` function, we memoize the result at each node, etc.

Once we have our Q-interpreter, all we need is a way of creating Q-programs, in the correct data format - as P-data. We can already create *pgraph*s from our textual format, so we complement this with a small translator from P to Q (written in Scheme).

4.2.7. Result of the Futamura projection

We can now perform the first Futamura projection, and we first do this on a program containing a single arithmetic operation:

```
(define (arith x y)
  (+ x y))
```

[3]Our Q-interpreter is available as `interp.pgraph` in our reference implementation - see Appendix C.

When we run our system on this example, within seconds the SMT process occupies all available system memory, we start paging to disk, and we never get anywhere. A single set of symbolic arguments can occupy several megabytes. Presumably, this is because our set comprehensions contain the entire computational history of all values, from the start of the program's execution.

We discuss and improve performance in more depth in 4.3, but an immediate adjustment is needed to observe a result for even this small example. We address this by applying our widening to all arguments, at all calls. This greatly simplifies the symbolic arguments to calls, and we are able to get a result. However, this means we are sacrificing accuracy: we are giving up relational information between arguments, across call boundaries. This is why we cannot perform the "cross-call" specialization mentioned in 4.2.4.

We view this as illustrating something important: discarding information (widening) is not just a theoretical consideration, for termination; it is also a practical consideration, for performance. This reinforces our impression from 3.4.5: choosing which information is kept and lost is a critical aspect of partial interpretation. Ideally, we would want a principled way to do this, which loses no accuracy, terminates theoretically, and performs well practically. It remains to be seen how well these ideals can be reconciled; we do not investigate different widenings, in this work.

Although we have just sacrificed accuracy in principle, the result we get here is excellent. Here is the output of our system, when specializing the Q-interpreter on the small arithmetic program:

```
(define (p0 x0 x1)
  (let ((t0 (+ (head x1) (head (tail x1)))))
    (cons t0 nil)))
```

This is *all that remains of the interpreter*: the two inputs are unpacked from the list of arguments (`x1`), added, and the sum is packed into a singleton list of results. The Q-program (`x0`) is no longer referenced, and all the internal logic and data structures of the interpreter have disappeared. In short, we have achieved a perfect first Futamura projection.

This confirms two things. First, our system is producing good results, even on a larger object program. Second, it does not require the object program (here, the interpreter) to be manually adjusted, in order for the specializer to produce good results.

This experiment takes 15s to execute; for a two-operation program, 45s; for a three-operation program, 1m40s. We consider this unacceptably slow, though we stress that it is not the specialization *of a two-operation program*, which takes 45s; it is the specialization of a relatively large interpreter, given a two-operation program to be interpreted.

In the next section, we analyze the performance of our system, and we show how its performance can be improved about a thousandfold. Conveniently, we can scale up the size of the Q-program as desired, to benchmark our increasingly efficient system.

4.3. Performance

In the last section, we saw that our system produces good results, but that it is slow. We now turn all our attention to the performance of partial interpretation and specialization. Our goal is to improve performance, but also to understand the design space for these improvements, and where they are most needed.

In 4.2.7, we already made a first modification to our algorithms, to improve their performance: we widen all arguments, at all calls. We will retain this modification, and in fact we will build on it to improve performance further.

Before we make more improvements however, we analyze the performance of our system. Any comments about experimental measurements refer to our running example of the first Futamura projection, as described in 4.2.

4.3.1. A starting point: three important quantities

We start from the theoretical picture put forth in 3.4. There, we showed that the complexity of partial interpretation depends on the number of cache entries, and the number of times their results are re-computed.

We also look at the practical picture: some straightforward measurements show that almost all our execution time is spent waiting for answers from the SMT solver.

Our first order of business is to reconcile these two apparently disconnected pictures. Instrumenting our code reveals that almost all our time is spent doing partial interpretation; not specialization. Specifically, almost all our time is spent checking set equality, as part of cache lookup, at *call* nodes.

In our theoretical analysis, execution time was formulated as the number of re-computations of the results of cache entries. We did not consider how long it takes to perform each such re-computation, because it was clear that it would terminate. However, we now see that handling a *call* node incurs a real practical cost - the bulk of *pgraph* partial interpretation.

This reconciles the theoretical and practical pictures: some number of cache entries are created; some number of re-computation jobs are run to compute results for those cache entries; and during these jobs, *call* nodes are encountered, where we perform cache lookup and query the SMT solver. We consolidate this into the following formula:

$$\text{execution time} = \text{total cache entries} * \text{jobs run for each} * \text{time to handle calls therein}$$

Next, we discuss what determines these three quantities, whether it's possible to decrease them, and how.

Cache entries are created for `(pgraph, args)` pairs. In a sense, the trace of `(pgraph, args)` pairs is a reflection of the program's execution path - a longer execution path means more `(pgraph, args)` pairs. This is not the static size of the program; rather, it is its "dynamic" size, when unrolled during execution. Intuitively, it is related to the program's run-time during ordinary execution.

In principle, cache entries are not necessarily created for *all* `(pgraph, args)` pairs: for given notions of widening and of argument equivalence, some `(pgraph, args)` pairs can get consolidated into the same cache entry. This is what happens when we have a cache hit.

Therefore the cache entries created are determined by the execution length of the program, and our choices of widening and argument equivalence in the cache. We have no control over the execution length of the program, and in this work we do not explore different widenings, or notions of argument equivalence. As such, in this work the number of cache entries is effectively something we observe, and have no control over. Experimentally, the number of cache entries is linear in the number of (arithmetic) operations in the Q-program, when performing a first Futamura projection.

The number of jobs run per cache entry depends on two things: avoiding re-computation altogether, and good worklist prioritization. We examine these two items.

In principle, it should be possible to avoid some re-computation by propagating congruence information between cache entries (see 3.4.4). Recall that this applies when two cache entries have arguments A and B, and results R_A and R_B, with $A \subseteq B$. We can then reason that $R_A \subseteq R_B$, also, and propagate R_A to R_B (i.e. set R_B to $R_A \cup R_B$). There is just one problem: experimentally, in all the cache entries created in these examples, we observe no pairs A and B with $A \subseteq B$. Therefore this idea won't help us here.

So far, we have not discussed the order of items in our global worklist. This is because the worklist order does not affect the correctness, termination, or worst-case complexity of our algorithms. However, the worklist order *can* affect the practical efficiency of partial interpretation. We have therefore made a small, intelligent effort in this direction, in our initial implementation.

We prioritize the re-computation of cache entries that have not been re-computed in a long time. To do so, in each cache entry, we store a timestamp indicating when the cache entry was last re-computed. Then, we order the worklist by prioritizing a smaller timestamp. When a cache entry is freshly created, its timestamp is initialized to zero - it is therefore prioritized in the worklist. When a re-computation finishes, we increment a global counter, and write the value of the counter as the result's timestamp. Intuitively, we are measuring time in terms of how many total re-computations have been performed, over the course of partial interpretation. This worklist priority scheme is due to [**KW94**].

In our experiments, under this worklist priority scheme, we are performing two computation jobs per cache entry. Intuitively, this is optimal: one job to create cache entries for the sub-calls whose results we need; and one more job to compute our own result, once the sub-results are available. Therefore there is nothing to gain by using a more elaborate worklist priority scheme - in these examples, at least.

We turn to the time it takes to handle *call*s. We first make a comment about the number of *call* nodes encountered. The object program dictates these, and with the way we are re-computing results (running the entire *pgraph* again), we need to handle all the *call* nodes during each re-computation. However, in principle it should be possible to track more precise dependencies between cache entries, and re-compute only part of the *pgraph* for a given re-computation job. For example, suppose A calls B, performs some post-processing on the results from B, and then returns something. If the result from B changes, clearly we need to re-compute the result of A. However, there should be no need to re-compute the *arguments* to B. We could therefore avoid the *call* node to B, if we re-computed only the data path from the result of B, to the result of A. We do not attempt this, in this work.

We focus instead on the time it takes to handle a given *call* node. We begin by reviewing what happens there. In the relevant situation, we have a unique *pgraph* being called. For that *pgraph*, we need to determine if there is an existing cache entry whose arguments are equal to the arguments we have now. We do this by traversing a list of cache entries, and checking set equality for each of them, using the SMT solver.

There are two issues with this. One issue is the order in which we perform these comparisons - the order in the list of existing cache entries. This determines how long it takes to get a cache hit. But the more serious issue is that when there *is* no suitable cache entry, we need to compare to *all* the existing entries, to rule them out. Then, we can create a new cache entry. This means that if we eventually have n cache entries for a given *pgraph*, it is inevitable that we have performed $O(n^2)$ comparisons to create them - the sum from 1 to $n-1$.

Note that there is no direct way to organize the cache entries, for example placing them in a binary search tree, or hashing their keys. As it stands, we have no notion of order on keys, and no way of constructing a hashing function; equality is only defined logically, via decision by the SMT solver. This reflects our theoretical development: we are completely reliant on the SMT solver.

Our most significant performance improvements will greatly reduce this reliance on the SMT solver. Before we discuss this however, we describe a simpler improvement. We omit minor improvements which will be made obsolete by our major improvements.

4.3.2. Ordinary execution

We refer to our first improvement as *ordinary execution*. When we encounter a *call* whose arguments are unique, we break out of the main partial interpretation logic. Instead, we invoke our ordinary interpreter with those unique arguments. When this completes, we write the (necessarily unique) results in a dummy cache entry, to later accomodate the passive mode used during specialization (see 4.1.1). This entry is not indexed in the cache; it is only saved at this call site (see 3.4.6). We then resume partial interpretation.

Ordinary execution achieves several things. To begin with, it avoids an immediate cache lookup, which would be costly. Beyond this, it avoids the subsequent partial interpretation of all sub-calls therein, and transitively so. Even better, it avoids the creation of cache entries for all these sub-calls, which keeps the cache smaller. This speeds up the handling of *call* in *other* partial interpretation jobs, which must handle *call*s by traversing the list of existing cache entries, performing costly SMT queries.

In our experiments, without ordinary execution, about half of all cache entries have unique arguments. With ordinary execution, we get a roughly twofold speedup.

4.3.3. Syntactically representing widened sets

Ordinary execution is a good thing, but it avoids the main problem: complete reliance on the SMT solver to perform cache lookup. We have not designed our set comprehensions to be analyzed by our own algorithms; they are essentially black boxes which we feed to the SMT solver, whose operation is also a black box.

It is time to change this. Our first step in this direction is the observation that all our arguments are now widened sets. Our plan is to exploit this structure, to improve the performance of cache comparisons.

As it stands, at the beginning of a *call*, we widen the current arguments. The widened arguments are then compared to the arguments in the existing cache entries, one by one. The key point here is that we are using the same representation (set comprehension), for both widened and unwidened sets. However, the class of widened sets is smaller, and a simpler representation is possible.

We start by defining a language of completely known values (VAL), in Figure 4.1 - this is nothing new.[4] We then define a language of widened sets (WIDEVAL), in Figure 4.2. WIDEVAL differs from VAL only by the added clause, *any*. *any* indicates a completely unknown fragment of a value, and can occur in the place of any WIDEVAL. Note that any expression in VAL is also an expression in WIDEVAL. This corresponds to the embedding of completely known values as singleton sets, in the partial information domain.

[4]Our type system still applies - see 2.1.3 and 2.5. For example, `cons` : α * (List α) $\rightarrow$ List α.

VAL ::=
  | INT
  | `false` | `true`
  | `mkProg` INT
  | `mkPair` VAL VAL
  | `nil`
  | `cons` VAL VAL

Fig. 4.1. VAL: our language of completely known values.

WIDEVAL ::=
  | INT
  | `false` | `true`
  | `mkProg` INT
  | `mkPair` WIDEVAL WIDEVAL
  | `nil`
  | `cons` WIDEVAL WIDEVAL
  | *any*

Fig. 4.2. WIDEVAL: our language of widened sets.

To ensure that WIDEVAL expressions are canonical, we introduce a single reduction rule:

$$\texttt{mkPair}\ \textit{any}\ \textit{any} \Rightarrow \textit{any}$$

We can therefore decide the equality of any two widened sets, as the syntactic equality of their WIDEVAL representations. Note that this generalizes to algebraic data types, with a similar reduction rule for any type which has a single constructor.

We modify our widening algorithm to target this WIDEVAL representation. The widening algorithm remains the same: using the SMT solver, we deconstruct a set comprehension into its known and unknown parts (see 3.4.5). The only difference is that we now construct a WIDEVAL, instead of a set comprehension. Where we would have represented an unknown fragment as an existentially quantified variable, we now write *any* instead.

We modify our cache entries so that their key is a WIDEVAL (along with a `pgraph`). A corresponding set comprehension is also needed during *pgraph* partial interpretation, to handle the *entry* node. This is easily constructed from the WIDEVAL: for each *any*, instantiate an existentially quantified variable.

Cache lookup can then be performed as follows. As before, we widen the current arguments, which is a costly SMT operation. However, once that is done, we have gained something substantial: we have a WIDEVAL, which can be compared *syntactically* to other WIDEVALs in existing cache entries. This is much more efficient than deciding set equality with the SMT solver.

This is a significant first step towards less reliance on the SMT solver, and it gets us another roughly twofold speedup. The bottleneck then becomes widening itself, at the beginning of a *call*. We next improve the performance of widening itself.

4.3.4. Disciplined set comprehensions

We want to improve the performance of widening, but it's not immediately clear how. Our widening algorithm itself seems fine: in general, we see no alternative to deconstructing the value by querying the SMT solver.

Instead of directly seeking to improve the widening algorithm, we can change our premises about its inputs. To choose good premises, we consider what we want to achieve in the output. The objective of widening is to identify all known "heads". A head is the part of a clause in VAL (see 4.3.3) which precedes any recursive occurrence of VAL. In other words, a head is the root of an expression. When we widen, we save the (uniquely) known heads, and we replace each non-uniquely known head by *any*.

It would be a good thing then, if known heads were readily available in our set comprehensions (the inputs to widening). We can achieve this by instituting a discipline (invariant) in *how* we use our set comprehensions: as much as possible, known heads will be written syntactically in the expressions of our set comprehensions. There will be no such thing anymore as "x such that $x = 1$"; write 1 then, and dispense with a variable and a proposition. Equivalently, the use of a variable will *imply a non-unique head.*

We explain our plan further, before we discuss how to implement it. This new discipline gains us several things. To begin, when we have a syntactically known head, we can trivially perform that step of a widening, without querying the SMT solver. Just as importantly, we now have a negative inference about variable occurrences. If we have x : List Int, in the absence of propositions we *know this widens to any.* This is because a variable implies a non-unique head, therefore it can be either `nil` or `cons`, which widens to *any*. And we will often *be* in the absence of propositions, *because* we have made an effort to move everything into expressions.

In other words, we are rebalancing our use of our set comprehensions - between expressions, declarations, and propositions. Previously, we prescribed no particular balance because our plan was to deliver all this to the SMT solver. For the SMT solver, it makes no difference whether something is 1, or x such that $x = 1$. Now, we aim for more expressions, and less declarations and propositions. We can *still* deliver this to the SMT solver, when we need to; but we will see that in many cases, we will not need to - at no loss in accuracy, and with a great gain in performance.

To maintain this new discipline (invariant), we must make three changes to how we perform *pgraph* partial interpretation: handling primitives, handling conditionals, and passing data between calls. We examine each.

First, we must change our handling of primitives (*prim* nodes): we will now perform computation over known heads. Previously, we handled a `+` operation by constructing a symbolic expression headed by $+$. Now, we first examine the operands to see if we can compute the sum ourselves (if they are both integer literals). When we cannot, we construct a symbolic expression, as we did previously. We handle `(head (cons _ _))` similarly - etc. This change is in line with our general plan: keep known heads readily available. Later we will want to widen, and if we see (+ ...), we don't want to start wondering what we know about that expression. If we see an expression headed by $+$, it must contain a variable (whose head is not uniquely known).

The equality checking primitive gets a special mention. Previously, as with all primitives, we handled `=` by constructing a symbolic expression (headed by $=$). Now, since known heads are readily available, we can perform some of the work of `=` ourselves, in two ways. On one hand, if we see two known and different heads, we can report `false`. On the other hand, if we see two known and identical heads, we can construct a conjunction of the equalities of the sub-expressions. For example, `(cons x nil)` and `(cons y nil)` are equal exactly when `x = y`. In the extreme case, we can report `true` (a vacuous conjunction) when both sides are syntactically equal.

Second, we must change our handling of conditionals (*if* nodes): we will now pull out common heads. The handling of *if* is already somewhat complex, but this change concerns the "main case", where none of the inputs are $\bot$ (the empty set), and the condition is undecided. In that case, previously we constructed an `(ite` c a b`)` expression. Now, we first examine the consequent and alternate branches, to see if they share a common head. If that is the case, we "pull out" the common head and push the conditional deeper inside.

This is best illustrated with an example. Suppose the consequent and alternate branches are `(cons 1 nil)` and `(cons 2 nil)`, respectively. These branches share a common `cons` head. We pull it out, and apply the same process recursively to the sub-expressions. Without this change, the end result would be `(ite` c `(cons 1 nil) (cons 2 nil))`; with this change, the end result is `(cons (ite` c `1 2) nil)`. Note that in this example, this process arguably simplifies the expression. This is not always true: if many sub-expressions differ, the duplication of `ite`s (and of the condition) can outweigh writing the common head only once. This is fine with us, because our objective is to make known heads readily available; not to simplify or complexify expressions.

Third, we must change how we pass arguments to calls. This involves the *entry* and *call* nodes. Previously, when we passed arguments to a call, we converted them to basic comprehension. The main feature of this conversion was that it introduced fresh variables for each argument or result. This policy would conflict with the discipline we are now introducing: if an argument has a known head, the new discipline forbids binding a variable to this argument.

We will therefore no longer convert arguments to basic comprehension. We can make this change, because the objective of this conversion has been subtly achieved in another way. The objective was to treat the arguments as a set, so that we could perform comparisons (set equality) with the SMT solver, as part of cache lookup. We have just replaced this with the conversion to the WIDEVAL representation (which expresses a set), followed by the syntactic comparison of WIDEVALs (in 4.3.3).

Note that this changes how we separate the caller's scope from the callee's scope. Previously, a "line" of fresh formals enacted this separation in a familiar way. Now, widening achieves a similar (but not identical) result: the separation becomes between *known and unknown*, instead of between *caller and callee*.

To understand the difference, suppose A calls B. Disregarding the details of naming, suppose A receives a completely unknown integer argument `x`. Suppose A now computes `(cons (+ x 1) nil)`, and passes this as an argument to B. Previously, we would have introduced a fresh variable, and propositionally asserted it equal to the entire argument: `(cons (+ x 1) nil)`. Now, we instead introduce a fresh variable for `(+ x 1)` only. This is where *any* gets written during widening, and *any* is then instantiated to a fresh variable. We therefore essentially pass `(cons y nil)`.

It may seem arbitrary that we are introducing variables for some parts of the arguments, but not others. We understand this "favoritism" as follows. We are preserving exactly the heads in the VAL language, which preserves our new discipline (invariant), allowing efficient widening. This is pragmatically a good thing, but we believe there is also something to be understood here. Notice that the only "operations" we are preserving are the algebraic constructors (here: `mkPair`, `nil`, and `cons`). We would argue that these are fundamentally different from the other, "actually computational" operations: `+`, `-`, etc. We explain why.

An arithmetic operation produces something new. This is tricky to state, for many reasons. Information theoretically, going from $2+3$ to 5 *loses information.* This is confirmed thermodynamically, because heat is dissipated when the sum is computed. Logically also, we "gain nothing" from having 5 rather than $2+3$, because 5 was already *latent* in the original information of $2+3$. Nevertheless, at the time when we have $2+3$, we don't have 5 *yet.* We need to perform some physical process to get from $2+3$ to 5. In this sense, an arithmetic calculation "does work".

In contrast, in some sense constructing (and accessing into) a data structure does *not* do work. It merely re-organizes what already exists.[5] We are used to thinking of (e.g.) `cons` as doing work, because its usual implementation places two things contiguously in memory. However, consider what happens if we dismiss this as an implementation detail. Algebraic construction (of data structures) then becomes "free" - it takes no "time". Therefore algebraic construction is not quite a computational *history*; it is a passive, "spatial" *arrangement* of data. And preserving this spatial arrangement coincides exactly with our needs for efficient widening. Even though our widening is chosen somewhat arbitrarily, this aspect of it - the special status of algebraic data structures - appears fundamental.

To some degree, the distinction we are making is not new. In the technical terms of λ-calculus, there is a distinction between *introduction forms* (constructors - here: `mkPair`, `nil`, and `cons`) and *elimination forms* (e.g. `+` and `-`, but also any accessor into a data structure, such as our `fst` and `snd`). We are suggesting a further distinction, between the "actually computational" eliminators (e.g. `+` and `-`) and the accessors (e.g. `fst` and `snd`). We leverage this further distinction in our widening, in the next section.

We end this subsection with a discussion of the modified behavior of *entry* and *call*. At *entry*, we instantiate the *any*s from our widened arguments, as fresh variables. We will give the consolidated behavior of *call* later, when we consolidate our entire algorithm. For now, we focus on how to consume the result from a cache entry.

Conceptually, we want to substitute "from the formals back to the actual arguments". However, now the variables in the formals do not correspond directly to the arguments; they correspond to the parts of the arguments which were not uniquely known. We therefore unify the formal parameters with the actual arguments. This gives us the correct substitution: from the variables in the formals, to the corresponding parts of the arguments.

This completes the changes needed to ensure our new discipline (invariant).

4.3.5. Improved widening

The discipline we have just instituted enables a much more efficient widening algorithm. We first explicitly define the language of expressions used in our set comprehensions (SYMVAL), in Figure 4.3. This will be the input to widening.

SYMVAL is the language we have already been using in our set comprehensions - essentially, the language of the SMT solver. Previously, we did not discuss this language much, because we did not manipulate its expressions much; we delivered them to the SMT solver. Now however, we will manipulate them more, and some comments are in order. Perhaps

[5]This is also tricky to state, because data can be encoded in the shape of a data structure. For example, a natural number can be encoded as the length of a list, as in Peano arithmetic. Nonetheless, we believe we are making a legitimate distinction.

```
SYMVAL ::=
    | INT
    | false | true
    | mkProg INT
    | mkPair SYMVAL SYMVAL
    | nil
    | cons SYMVAL SYMVAL
    | VAR
    | ACCESSOR SYMVAL
    | PRIM SYMVAL ...
    | ite SYMVAL SYMVAL SYMVAL

ACCESSOR ::= fst | snd | head | tail

PRIM ::= + | - | * | div | = | < | <= | > | >= | and | or | not
```

Fig. 4.3. SYMVAL: our language of expressions in set comprehension.

the most important is that although this is starting to look strangely like a programming language, we stress that this is a language for *symbolic expressions*. We clarify the distinction.

As in a programming language, there are reduction rules over this language. These correspond to computation over known heads, as introduced in 4.3.4. However, unlike evaluation in a programming language, it is likely that reduction will "block", for example at VAR + INT. Here variables are not simply "bound to a value", as in programming; they are more like mathematical variables (unknowns). Another distinction is that there is no CALL here. So we can only express intra-call computation, which is exactly what we need. We already have other machinery to handle calls.

The other important point is that we continue to distinguish accessors from "computational" primitives - as we have started to in 4.3.4. These two cases are handled separately in our improved widening algorithm below.

Note that any expression in WIDEVAL (see 4.3.3) is naturally included in SYMVAL, by instantiating the *any*s as VARs. This corresponds to the inclusion of our approximation lattice (of widened sets), in our larger lattice of set comprehensions.

We now describe our improved widening algorithm. Previously, widening was type-directed; now, it is syntax-directed. We explain how to handle each clause in SYMVAL.

- Known integers, booleans, and programs widen to themselves.
- Known algebraic constructors (here: `mkPair`, `nil`, and `cons`) are applied to their (recursively) widened sub-expressions. After we widen the sub-expressions of `mkPair`, if possible we apply the reduction rule: `mkPair` *any any* ⇒ *any*.
- In the VAR case, we check if there are any propositions in scope - there are often none. Without propositions, any VAR widens immediately to *any*. This is justified by our discipline (invariant): any variable reference implies a non-unique head.

In the presence of propositions, we fall back to our old widening algorithm, which deconstructs this variable by querying the SMT solver.

- In the ACCESSOR case, it is remarkable that we can apply the same logic as for VAR: in the absence of propositions, widen to *any*. This is justified by a combination of the changes we have introduced. We cannot have a constructor beyond an ACCESSOR, or we would have already reduced this syntactically. The only thing we can have is VAR or `ite` - possibly preceded by a chain of accessors. As any VAR has an unknown *head*, we can't possibly have information about one of its *components*. Similarly, an `ite` conceptually has an unknown head, because we have already pulled common heads out of `ite` (see 4.3.4).
- In the PRIM case, we always check for uniqueness with the SMT solver. We do this to avoid a loss of accuracy: an arithmetic expression can have a unique value by mathematical identity, even if it contains a completely unknown variable.
- In the `ite` case, we proceed as for VAR: in the absence of propositions, widen to *any*; otherwise deconstruct with SMT.

To maximize the value of this effort, we memoize the results of widening. We do this by hashing the symbolic expressions (SYMVALs) that we widen, within each *pgraph* partial interpretation. We preserve these memoized widening results for the later specialization phase: uniqueness is easy to decide given a widened value. We then memoize further uniqueness results during specialization itself. Applying all this gets us an overall hundredfold speedup, on top of previous gains. We then introduce two minor improvements: we reset a single SMT process for every job, and we hash WIDEVALs for cache lookup.

Appendix B shows a final consolidated view of partial interpretation, including all the changes made in this chapter.

4.3.6. Final picture of performance

With the performance improvements we have made, the system runs about a thousand times faster than it did originally. Taking the Futamura projection of the three operation program took 1m40s (100s); now, it takes 0.15s. This gap grows for larger examples; it is slightly smaller for the smaller examples. About 80% of total time is spent on partial interpretation; the rest, on specialization.

It is now possible to perform the Futamura projection of a 120 operation program, in about two minutes. We stress again that this is not the specialization *of a 120 operation program* (this would be instantaneous); this is the specialization of a relatively large *interpreter*, which is running a 120 operation program.

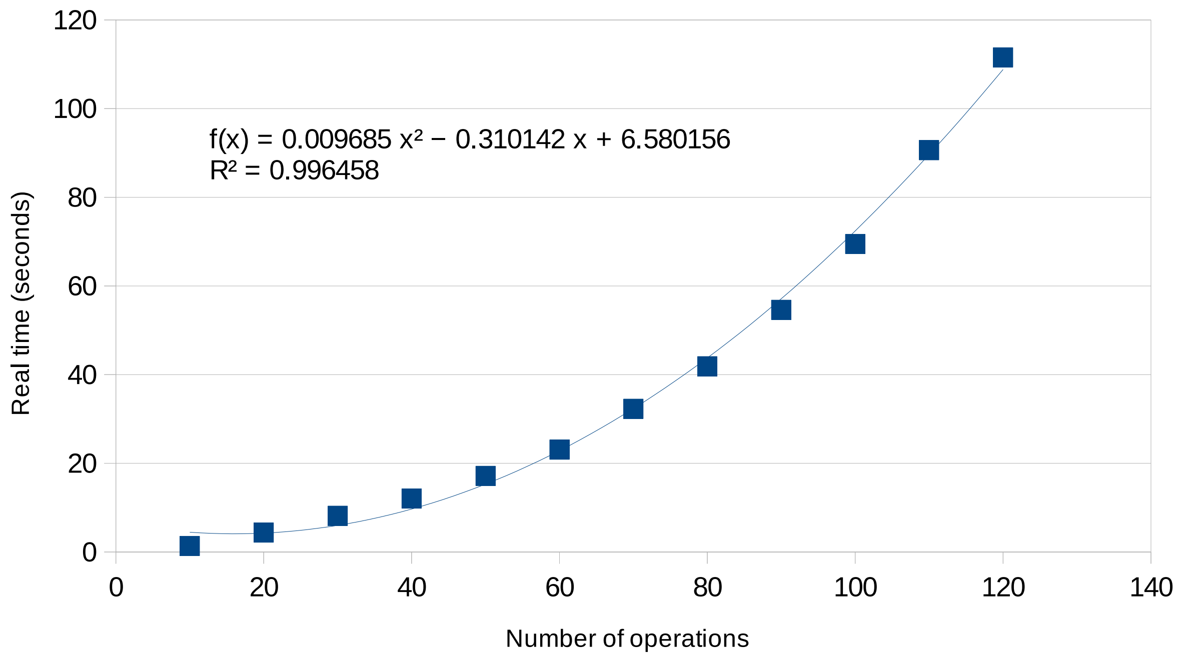

Fig. 4.4. Time spent performing the first Futamura projection, for a growing input program.

Figure 4.4 shows the growth of execution time, for a growing interpreted program, in increments of 10 operations. Measurements were made on an Intel i5-6600, running Linux 5.12. Our code was compiled with version 4.9.3 of Gambit Scheme [**Fee19**]. The Q-programs to be interpreted were generated algorithmically, using a combination of additions and multiplications. The SMT solver was CVC4 1.8, set to timeout with UNKNOWN after 20ms for each query. See Appendix C for information on reproducing our results.

The growth is quadratic in what we could call the *problem size*: the execution length of the object program. Partial interpretation gets results by running a program, from start to finish. Here, the object program is the interpreter, and this interpreter runs longer if it must interpret a larger program (from 10 to 120 operations).

Originally (in 4.3.1), we stated that almost all the execution time was spent waiting for answers from the SMT solver. Our considerable improvements have changed this performance profile. Figure 4.5 shows where time is spent now.

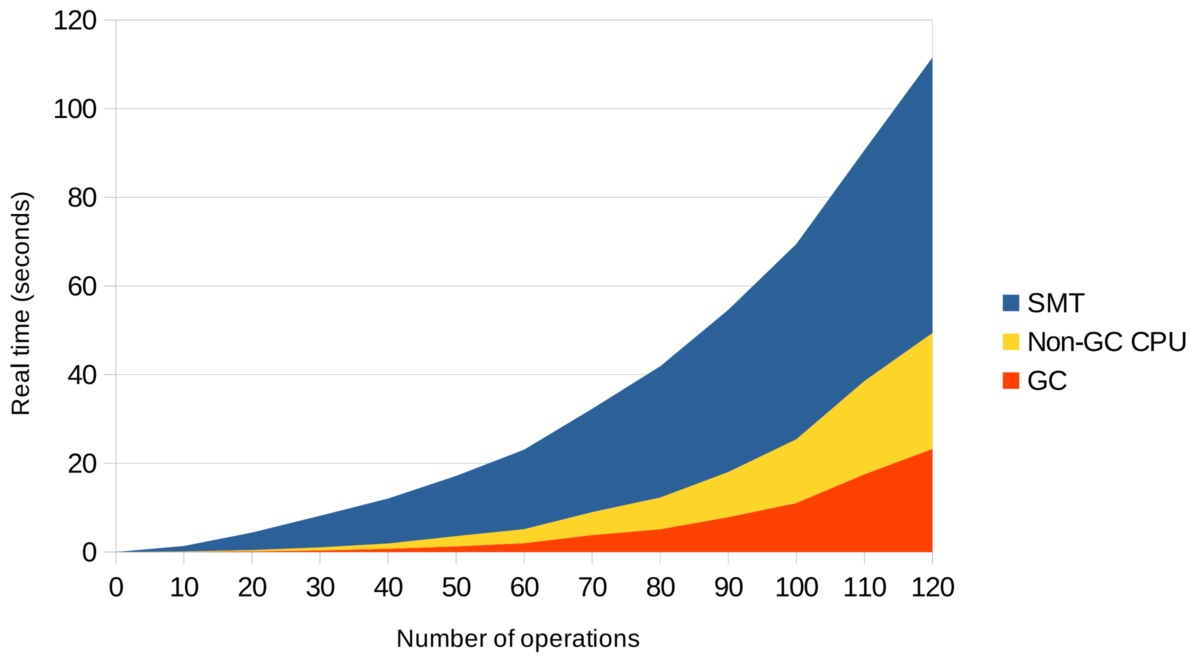

Fig. 4.5. Breakdown of time spent performing the first Futamura projection.

About half the time is still spent waiting for the SMT solver: checking uniqueness, during widening or during specialization. Aside from that, we see two new significant items. First, garbage collection takes up about a quarter of our execution time. Second, our own algorithms ("non-GC CPU") also take up about a quarter. Of this quarter, about 80% is spent doing widening; about 20%, performing syntactic cache comparisons. Each of these three items appears quadratic in the problem size - measured experimentally. And as far as we can tell, they are the only quadratic items. In other words, we have done enough initial optimization to identify significant bottlenecks.

It would take further work to understand exactly why these growths are quadratic. In our experience, truly figuring that out is done by trying to find a better way. Nevertheless, we offer some insight.

Fundamentally, these three growths appear connected: the *number of widening steps* is quadratic. Some of these steps are performed "in-house" (by our algorithms); some of these steps are performed by the SMT solver. And as we do all this, we allocate a lot of memory, which is likely why the garbage collector is doing so much. Memory usage is also growing quadratically: a peak of about 1.2 gigabyte, in our larger examples - measured experimentally with Unix system tools.

We make the obvious comment that in an implementation language with manual memory management (such as `C` or `C++`), the garbage collector would not be a problem like this. Most of our allocations have a clear and limited lifetime, and we could leverage this to optimize memory management.

Nevertheless, it seems that the key parameter is the number of widening steps. Intuitively, it appears to be quadratic because:

- as the problem grows, we have more cache entries, more jobs to compute their results, and more *call*s therein;
- as the problem grows, additionally the arguments to calls themselves grow, so it takes longer to widen them at each *call*.

This might be a feature of Futamura projections, or it might be a general fact about our partial interpretation algorithms. Further investigation is needed, but we leave it to future work.

4.3.7. Possible further performance improvement

We mention one idea beyond using manual memory management. One important source of SMT uniqueness queries comes up as follows.

In our widening algorithm (in 4.3.5), we defer to the SMT solver to establish the uniqueness of arithmetic expressions. This is because in the broadest sense, "anything could be unique, by mathematical identity" - we can't tell... or can we?

Suppose we have a large arithmetic expression which depends only on two variables - this is what happens in our examples. Conceptually, it seems possible to efficiently decide the uniqueness of all sub-expressions, as follows.

First, obtain two models *just for the two variables*: two pairs of values that satisfy whatever constraints are in scope. In general, this can be done by querying the SMT solver. In the absence of propositions, we can even select two models ourselves, because there are no constraints on the variables.

Next, begin ordinary evaluation of the large arithmetic expression, *over the two models at once*. For each sub-expression, compute two values (one for each of the two models). Now compare the two values:

- if they are different, we have determined that this sub-expression does not have a unique value
- if they are the same, we can expend an SMT query to verify whether they are *always* the same (over all possible models).

We speculate that refining this idea would further improve performance, substantially.

Chapter 5

Conclusion

5.1. Related work

5.1.1. Berlin, Weise, Ruf, Katz, and FUSE

Our work is perhaps most influenced by a particular line of work on partial evaluation. This line of work was mostly carried out in the early 1990s, at Stanford and MIT, under the broad name FUSE. We review it chronologically.

In 1989 at MIT, Andrew Berlin completes his masters project [**Ber89**], advised by Sussman. Berlin performs the partial evaluation of a functional subset of Scheme. He specializes straight-line code and conditionals, but not recursive calls, which are handled separately. Correspondingly, he applies this to scientific (numerical) computations, which involve a lot of straight-line code. This is reported as very successful, especially targeting supercomputers.

Berlin's system performs a kind of symbolic execution of the program, and is therefore a precursor to our work. In particular, one of the most important aspects of the Berlin partial evaluator is the construction of (and access into) symbolic data structures of known shape. Berlin also uses graphs to represent specialized programs; but not source programs.

At the time, Daniel Weise is a professor at Stanford, and a recent PhD graduate from MIT, also advised by Sussman. He begins expanding on Berlin's work, under the working name FUSE. An important milestone is [**WCRS91**], which shows for the first time how to specialize recursive calls automatically.[1]

[**WCRS91**] introduces the idea of tolerating non-termination of the partial evaluator, when the object program would not terminate at run-time; the guarded recursion condition; and the union with previous arguments. We use these as our particular termination goal, assessment, and resolution, in 3.4.6.

[1]For an *online* partial evaluator - see 5.1.3 for a brief discussion of what that means.

At the time, Erik Ruf is a PhD student at Stanford, advised by Weise. Ruf also begins expanding on the work above. He makes several contributions relevant to our work. These (and others, less relevant to us) are consolidated in his PhD dissertation: [**Ruf93**].

Ruf introduces the general architecture we use, with a cache, a worklist, and a fixed point computation. He does this to compute more accurate results from sub-calls; previous work considered the results of sub-calls to be completely unknown, simply. We consider this a first trace of what we call partial interpretation.

Unlike us, Ruf does not separate specialization and partial interpretation, algorithmically or conceptually. They are performed simultaneously: in a single phase, a fixed point is found, not only over the results of calls, but also over specializations. The results of calls are viewed as secondary, serving to improve the (primary) accuracy of specialization.

Ruf's efforts are greatly complicated by the perceived necessity of handling closures. This reflects the lineage of his work, back to Berlin: the partial evaluation of Scheme. In contrast, we do not require our work to apply to an existing programming language. We allow programming language design to proceed cooperatively with the design of partial intepretation, and specialization. We avoid closures and their complications, entirely.

Ruf also introduces some of the pieces of our termination argument. In particular, he identifies the technical problem we study in 3.4.4. He suggests a solution which we believe is mathematically equivalent to our "second approach": propagating all congruence information. However, he applies his solution only during cache lookup, and he does not leverage congruence to write into the cache. He does not appear to recognize congruence as an opportunity to reach a fixed point faster.

Ruf also introduces a notion of argument equivalence which is broader than exact equality. He does this by 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 methods, and we discuss it again in 5.3.

At the time, Morry Katz is another graduate student at Stanford, and he also begins expanding on the work above. In [**KW92**], Katz and Weise clearly define a lattice which we adapt as our particular approximation lattice (see 3.4.5). The previous works use a similar lattice implicitly. Note that in all these works, there is no notion of *widening* to that lattice: that lattice exists alone, without being included in a larger lattice (such as our partial information domain).

Then, Katz's PhD dissertation [**Kat98**] proposes several new ideas. We find these interesting, but we are not sure what their proper place is. We do not mirror Katz's presentation, but we identify three main ideas.

First, Katz suggests that we can expand on Ruf's tracking of information use. In addition to tracking the information used by *specialization*, Katz essentially suggests that we can track the information used by (what we call) *partial interpretation*. These two aspects of information use are never disentangled, in his treatment.

In the context of partial interpretation, in order to compute what information is used in the arguments, we must have a notion of what information is needed in the results. This is Katz's second idea: calls can be indexed not just by a (program, args) pair; there can additionally be a third piece which indicates what information is needed in the result.

Katz suggests that this gives us a way of determining what information is relevant, at all call and return boundaries. We think this is a powerful idea, because that is exactly what we would need to do "adaptive widening", as suggested in 3.4.5.

However, Katz then suggests (this is his third, and arguably main idea) that we can use this to assess the *termination* of partial interpretation/evaluation. He abandons guarded recursion (see 3.4.6), and suggests that we can let the cache get populated without intervention. He argues that tracking information use induces equivalence classes of arguments, and that termination will be ensured naturally by cache hits over these equivalence classes.

Katz shows that this works, but that it is not satisfactory. It is too conservative: it terminates too soon. He introduces a heuristic (*base case analysis*) to make it less conservative. Then, once again his algorithms do not terminate, in certain cases.

Our assessment of this is that information use is not necessarily the right idea to assess *termination*. Intuitively, it seems that we only know what information is used, *after* we have taken a stand on termination (and possibly intervened to ensure it). However, we do think that information use has a place in future work on partial interpretation.

From our perspective, this line of work was almost exactly in the right direction (of more accurate partial evaluation). Despite this, to the best of our knowledge this line of work remains uncontinued - until now, in some sense. Weise and Ruf went to Microsoft Research. There, Weise jointly contributed (among other things) the worklist priority scheme [**KW94**] that we use (see 4.3.1). We do not know what happened to Berlin or Katz.

5.1.2. Symbolic execution and abstract interpretation

We do not have the expertise to give a deep comparison to symbolic execution (e.g. [**CGP$^+$08**]) or abstract interpretation (the original reference is [**CC77**]). Nevertheless, our work combines ideas from each.

As in symbolic execution, we use symbolic formulas to represent data values. However, symbolic execution traditionally executes only a fraction of the program, to avoid termination problems. Instead, we execute the program completely, and we directly address the termination problems that arise.

To address these termination problems, a key piece in our work is the conjectured lossless abstraction at a call, from a computational history to a set (see 3.3.1). This lets us apply ideas from abstract interpretation: fixed point computation with a cache and a worklist. We complete this with widening (also an idea from abstract interpretation), and termination assessment/resolution. We adapt our choice of these from the work reviewed in 5.1.1, and this gives good results for partial evaluation. However, this choice is not principled.

We believe a principled choice could be possible. We have mentioned the idea of "adaptive widening" (see 3.4.5), which would widen differently at different execution points, keeping only relevant information. This possibility appears related to some recent developments in abstract interpretation, under the name *a posteriori soundness* [**MM09**].

As for termination assessment, Nguyen [**Ngu19**] has recently used an SMT solver for symbolic execution, especially applied to gradual typing [**MNTHVH21**]. Significantly for us, Nguyen uses *size-change graphs* (e.g. [**BAL07**]) to assess the termination of execution paths. This strikes us as a potential enhancement to our methods, replacing guarded recursion as the termination assessment in our general termination framework (see 3.4.6).

5.1.3. Other work on partial evaluation

The FUSE line of work (see 5.1.1), and our own work, stand apart from most work on partial evaluation. We discuss the differences briefly.

Historically, partial evaluators have fallen into two camps: *online* and *offline*. FUSE and our work are online partial evaluators: they decide how to specialize the program using all available information about data values. In contrast, most other partial evaluators are offline: they decide how to specialize the program in advance, based only on an estimate of which values will be known vs unknown. [**JGS93**] covers many topics in offline partial evaluation.

Offline methods make a sacrifice in accuracy.[2] In return, they achieve faster specialization, and maneuvers such as self-specialization and higher Futamura projections (see e.g. [**Glü09**]).

We view offline methods as complementary to our work. We focus on accuracy, and this is a challenging problem in itself. However, in principle there is nothing preventing a hybrid approach: decide all you can in advance, but not when this might sacrifice accuracy. We speculate that this could reconcile the benefits of the online and offline approaches.

We note [**Sah91**] and [**BLR09**] as other examples of online partial evaluation, for the Prolog language. We have only passing familiarity with these works.

[2]Chapter 2 of [**Ruf93**] argues this convincingly.

5.1.4. Program representation

Our *pgraph* representation has the flavor of a compiler IR (intermediate representation). We compare it to other such representations:

- SSA: static single assignment [**RWZ88**]
- VDGs: value dependence graphs [**WCES94**]
- PEGs: program expression graphs[3] [**TSTL09**].

Though our *if* node is a simple conditional expression, we also view it as the logical conclusion of a path started upon by SSA representations. A *pgraph* is factually "static single assignment", since it is pure. But we believe this is not the crux of the matter in SSA.

The notion of SSA begins with a CFG (control flow graph), which totally orders execution. Then, an SSA transformation is performed, which renders explicit the connections between uses and definitions. These use-def connections are effectively data dependencies, and this is in our view the central feature of SSA: a data dependency graph *superposed over* a CFG.

The data dependencies alone are enough to characterize many data values. However, some values are selected based on control flow, which is an issue in the pure data-dependency view. These cases are expressed in SSA with the Φ node, which records two values to be selected between, and implicit information about which to select depending on which control path was taken. Therefore Φ is in a sense a bridge between the two aspects in SSA: it indicates a data dependency, conditioned on some information coming from the CFG.

In contrast, our *if* node records the explicit boolean condition under which to select the values. This is simpler to analyze, which is our primary goal; but this is not the only consequence. Since the conditionality no longer depends on control information, the control aspect of SSA becomes superfluous and can be removed. To be more precise, the control aspect is *minimized*; we can never *eliminate* control, which is order of execution. In the original CFG there is a total ordering on execution, and some of it remains relevant in SSA; in a *pgraph*, execution is partially (and minimally) ordered - by the data dependencies.

*pgraph*s are similar to VDGs (value dependence graphs). The two representations are distinct primarily because we have no built-in notion of closure, and because we inject different primitive data and operations.

*pgraph*s are also similar to PEGs (program expression graphs), in that our *if* node records an explicit boolean condition. However, PEGs include imperative looping constructs, whereas we rely on recursion to express repetition.

[3]Not to be confused with *parsing expression grammars*.

5.1.5. Pypy and Truffle/Graal

Pypy [**RP06**] and Truffle/Graal [**WWH$^+$17**] are recent implementations of highly dynamic languages (such as JavaScript, Python, and Ruby). Although the aim of these works is very different from ours, they use some related techniques. To understand these works, we must briefly examine the history of compilers, programming languages, and personal computers.

In the early days of compilers (roughly 1960-1990), the main concerns were performance of the compiled program, and ensuring that compilation itself was tractable. These concerns reflected the limited computational resources that were available (hardware). As a result, programming languages were designed *for the compiler*; not for the programmer. The discipline of e.g. C and FORTRAN makes the compiler's job easier.

Between about 1990 and 2005, hardware leapt forward by several orders of magnitude. Suddenly there was a world of possible, useful programs, which could run *without* hitting the wall of hardware performance. All we needed was... the programs. As a result, there emerged programming languages aimed at rapid development: offering great flexibility to the programmer, and intending to be interpreted, not compiled. JavaScript, Python, and Ruby are such languages which became popular, and large code bases exist in these languages.

After about 2005, the increase in hardware computing power slowed radically. It became increasingly relevant for programs to perform better, in particular in highly dynamic languages. In other words, it became relevant to *compile* highly dynamic languages to efficient machine code, even though these languages were not designed for such compilation. Pypy and Truffle/Graal aim to fulfill this need. This is completely opposed to *our* aim, which is to design the programming language cooperatively with the needs of automation in the compiler: partial interpretation and specialization.

Pypy focuses on Python. Conceptually, Pypy first takes a stand that a certain subset of Python both includes the most important features of Python, and is sufficiently disciplined to be successfully compiled. This subset is called RPython, and it embodies object-oriented programming with statically inferred types. RPython effectively becomes a compiler IR, and the plan is then in two parts: first, transform a Python program into RPython (in a *frontend*); then, compile RPython to efficient machine code (in a *backend*).[4]

Translating Python to RPython is not easy, because by premise full Python contains undisciplined features which do not map directly to RPython. Pypy specifies this translation by providing an interpreter for full Python, written in RPython. We have seen that defining an interpreter induces a translation, via the Futamura projection. Pypy leverages the fact that a translation is induced, but not using the Futamura projection. We have

[4]Object-oriented runtimes are also discussed as backend targets: .NET and Smalltalk.

also seen that an interpreter is conceptually split in two parts: “decoding” the source program, and “performing” corresponding actions in the host environment. Pypy manifests this separation architecturally in the interpreter: the interpreter does not perform any actions directly; instead, the interpreter invokes a separate black box module when it wishes to perform actions. Instantiating this black box to actually perform actions yields an ordinary interpreter. Alternatively, instantiating this black box to construct RPython IR yields a translator.

To translate from RPython to C (and then machine code), Pypy first performs abstract interpretation[5] with a value domain designed for the specific needs of Python. The objective of this abstract interpretation is to limit the possibilities for dynamic dispatch at run-time. Any remaining possibilities are monomorphized into regular C code, and some form of garbage collection is introduced.

Truffle/Graal has the same overall architecture as Pypy:

- an IR which captures the most important features of dynamic languages, but is sufficiently disciplined to be compiled
- a frontend from dynamic languages to this IR
- a backend from this IR to efficient machine code.

The IR used here is Java byte code, as used in the Java Virtual Machine, with a few modifications. This choice reflects the origin of the work on Truffle/Graal: Oracle Corporation, which previously acquired Sun Microsystems, which in turn had developed Java. The name of the frontend is Truffle; the name of the backend is Graal.

Like Pypy, Truffle specifies the frontend translation as an interpreter written in their IR (modified Java). Unlike Pypy, Truffle explicitly intends to support several dynamic languages, and interpreters are provided for (at least) JavaScript, Ruby, and R. Also unlike Pypy, Truffle performs the first Futamura projection: translation by specialization of the interpreter. Their partial evaluation is online and reminiscent of pre-Weise work: explicit annotations are given in the interpreter to limit the unrolling of loops and calls during partial evaluation. These annotations account for the modifications made to Java byte code.

Graal uses a lower-level internal representation of programs as graphs. These graphs are of the SSA kind described in the last section: they combine data dependencies and control flow. Specifically, Graal’s graphs descend from the work by Cliff Click [**CP95**] at Sun Microsystems, originally used in the HotSpotVM Java implementation.

Both Pypy and Truffle/Graal face additional difficulties because they are just-in-time (JIT) compilers. These additional difficulties are not directly relevant to this work.

[5]The literature on Pypy often refers to this step as “type inference”.

5.1.6. Formal systems

It is interesting to briefly compare the language of propositions used in our set comprehensions, to the languages used in formal systems with dependent types, such as Coq [**HKPM97**] or Agda [**BDN09**]. We do not have recursion in our propositions, while these formal systems do. But the distinction is deeper than that: we can express sets which in these systems would only be expressible with dependent types (e.g. pairs of integers, the second of which is twice the first).

In these formal systems, proofs and propositions are first-class objects: they have a type, within the formal system. This is usually achieved through a Curry-Howard embedding of logic, where propositions become types. In our (SMT) setting, neither proofs nor propositions are first-class objects (they do not have an SMT type), and propositions are not types. Therefore our propositions can be expressive without dependent types. This is similar in spirit to *refinement types* [**FP91**], but we do not claim a comparison.

In some sense, our setting is based on a different "correspondence" than the Curry-Howard correspondence used in (e.g.) Coq or Agda. Under the Curry-Howard correspondence, propositions correspond to types. For us (and for SMT solvers), the correspondence is between propositions and boolean expressions.

5.1.7. Automatic deduction

We have focused on SMT solvers as an example of modern automatic deduction, but automated theorem provers (e.g. [**KV13**]) are an alternative. We also mention the variants of natural deduction discussed in [**HR04**].

By extension, we can also view as automatic deduction anything which automatically:

- simplifies symbolic expressions
- demonstrates their equivalence
- manipulates them in any way towards a useful goal.

We therefore mention modern program optimization (e.g. equality saturation [**TSTL09**]), and the subject of computer algebra.

We mention these alternatives because our choice of an SMT solver imposes constraints on our system. In particular, we have no recursion in our propositions, we inherit SMT's traditional weakness with universal quantifiers, and we face a substantial performance challenge when we query the SMT solver. By replacing or augmenting the SMT solver with an alternative form of automatic deduction, it may be possible to alleviate these difficulties. We revisit this idea in 5.3.

5.2. Discussion

At the beginning of this text, we set out to develop a principled approach to accurate partial evaluation. We believe we have taken a step towards this. Specifically, we have:

- clearly separated partial evaluation into specialization and partial interpretation (computation over partial information)
- walked through how to do both, in theory and in practice
- formulated partial information to ground it in modern automatic deduction (specifically, an SMT solver)
- shown how to reduce reliance on the SMT solver, to improve performance by three orders of magnitude.

Our implementation has reproduced basic results in partial evaluation, without requiring manual adjustments to input programs. Additionally, we have achieved a gain in intra-call accuracy, over previous partial evaluators. And this is all *despite* the numerous limitations in our implementation.

We inventory these limitations in the next section, but we view them as *opportunities*. They can conceivably be addressed, and our system can then handle more difficult problems than the rudimentary examples we have experimented on.

This suggests the following plan. First, address the most pressing limitations on object programs. Then, start experimenting on more serious object programs and libraries. Every time the results are good, we are one step closer to mainstream partial evaluation.

Every time the results are poor, we have an example failure. The next step is key: *diagnose* the failure in terms of the limitations in the next section. This directs our work to address those limitations first. More importantly, our approach *remains principled*, even though it is guided by practical needs.

It seems inevitable to eventually achieve one or both of the following:

- sufficient accuracy for mainstream partial evaluation
- a theoretical understanding of what about this is difficult, or impossible.

5.3. Limitations and future work

These limitations are of two kinds. In some cases, the right solution seems clear, but we just haven't delivered it. In other cases, we don't know what the right solution is; we have then delivered arbitrary solutions, usually adapted from past work.

5.3.1. Object programs

Our object programs (*pgraph*s) lack full algebraic data types (see 2.5), and some primitive data (see 2.1.3): machine words, and strings. These are important practical additions, and we anticipate no difficulty with this.

Our object programs also lack many features, such as: external effects, mutation, convenient looping, non-local exits (e.g. continuations). These require fundamental changes to *pgraph*s. Several existing programming languages offer such constructs. However, it might be a mistake to rush to adapt existing treatments to our work. The challenge is to handle these constructs in a way which best integrates with partial interpretation.

5.3.2. Theoretical considerations

In 3.4.4, we studied the question of congruence of cache entries, but we did not settle on a solution. Many questions remain:

- exhibiting a real program on which partial interpretation fails to terminate, due to the problem discussed in 3.4.4[6]
- determining whether our two suggested approaches produce the same end result.

If our first and second approaches produce the same end result, then we believe either approach is right theoretically. The choice would then depend on performance considerations.

Also, in 3.3.1 we left a conjecture informal and unproven, about the irrelevance of histories.

5.3.3. Accuracy of partial interpretation

We have limited the accuracy of partial interpretation in many ways. First, we are discarding information (widening - see 3.4.5 and 4.2.7) at all call and return boundaries. We have seen that this is theoretically necessary in some places, for termination; but also practically desirable everywhere, for performance. However, currently this is done arbitrarily.

Instead, we believe there should be a principled way of determining which information is relevant or not, at each boundary. This would possibly expand on ideas from [**Kat98**]. We believe this is a critical aspect of future work on partial interpretation.

[6][**Ruf93**]'s section 4.4.2 should be consulted before undertaking this, because Ruf gets part of the way there.

Second, we have seen (in 3.4.6) that we need to limit the number of cache entries, to ensure termination. Currently this is done by assessing termination arbitrarily. Additionally, we tolerate non-termination of partial interpretation, when the object program might also not terminate (at run-time).

Instead, we believe there should be a (more) principled way of assessing termination, such as size-change graphs (e.g. [**BAL07**]) as used by [**Ngu19**]. We also mention techniques based on linear algebra: [**Tiw04**].

Third, in the language of propositions for our set comprehensions (see 3.1.1), recursion is excluded. This is done to match the perceived requirement of current automatic deduction. We do not have the expertise to assess how flexible this requirement is.

Fourth, we do not track path conditions (see 3.2.7). We anticipate no theoretical difficulty with this, but it may require some re-architecture of our implementation.

Fifth, we make no effort to analyze calls when the called program is not uniquely known (see 3.2.5). We anticipate no serious difficulty with this.

5.3.4. Performance of partial interpretation

There are many opportunities to improve the performance of our methods. First, we return to our notion of argument equality from 3.3.2. We said that we require equal monomorphic types for argument equality in the cache.

However, consider a polymorphic list length program, applied to two inputs:

- `[x, y, z]`, where each entry is a completely unknown *integer*
- `[x, y, z]`, where each entry is a completely unknown *boolean.*

Ideally, we would like both of these calls to fall in the same cache entry. As a slightly less obvious example, `[x, 2, 3]` "should" also fall in the same cache entry.

Clearly there is an opportunity here for a broader notion of argument equivalence, and polymorphism would play a role in this broader notion. The result of a program cannot depend on either the type or the specific value of a polymorphic argument - this is what it means for a program to be (parametrically) polymorphic.

We do not know how to formulate this broader notion of equivalence. However, this appears related to Ruf's work on broader equivalence of arguments, in [**Ruf93**]. All this is especially significant, since the number of cache entries is a primary parameter in the performance of our algorithms.

Second, in 4.3 we introduced some optimizations which reduce our reliance on the SMT solver. In 4.3.7, we suggested a further improvement in that direction. But if the path to performance is reduced reliance on the SMT solver, at some point we have to ask whether the SMT solver is the right tool for us.

In hindsight, this question is not so surprising, because the SMT solver is designed for a slightly different problem than ours: deciding satisfiability, and constructing a model. In contrast, we usually already *know* that our queries are satisfiable, because they constrain data which is the result of executing a program. And we often don't care to have a particular model, if there exist two or more models - then the data is non-unique.

We still think it's right to ground our work in *modern automatic deduction*; just perhaps not an SMT solver. We have mentioned some alternatives, in 5.1.7. It's also possible that partial interpretation requires a new variant of automatic deduction. In that case, our work in 4.3 can be seen as its starting point - especially 4.3.3 to 4.3.5.

Third, in 4.3.1 we mentioned the possibility of not re-computing an entire *pgraph*, for each re-computation job. We anticipate this to be a technical challenge, but no more.

Fourth, we mention the opportunity for parallelism in our methods. Parallelism appears naturally at the level of re-computation jobs, but also at the level of queries to SMT (or another form of automatic deduction).

In our examples, we have not observed a real opportunity for job-level parallelism, during *partial interpretation*: the worklist is very short at any given time. However, we have observed long worklists during *specialization*.

5.3.5. Accuracy of specialization

We have seen (in 4.2.1) that "checking for uniqueness" does not account for all of program specialization/optimization. This should be supplemented with (ideally) principled program optimization. Equality saturation [**TSTL09**] seems especially relevant.

5.3.6. Formulation as inference rules

The committee has suggested that we formulate our methods as logical inference rules. We think this idea is potentially very strong, but we are unable to deliver on it at this time. We take a moment to give our interpretation of this idea, and to comment on its difficulty.

Our methods combine many pieces: automatic deduction as a black box, our partial interpretation algorithms, our formulation of specialization, and we're talking about supplementing all this with principled program optimization. By formulating all these pieces in a common language of logical inference rules, we could hope to find a unified and simplified view of our methods, removed from implementation details. Any overlap between the pieces is an opportunity for unification and simplification. For example, equality saturation is used in program optimization, but also to implement congruence closure within modern tools for automatic deduction.

Such a unified formulation is a kind of formalization, and as such could facilitate proofs. But in our view, the real win would be to study the core so formulated, and to match it to the strengths and weaknesses of different approaches to automatic deduction - either existing or our own. We could then select the automatic deduction best suited to the needs of partial interpretation and specialization. Intuitively, instead of trying to fix many comparatively minor problems in our system, we would change the core of the system itself (deduction); then we would see what minor problems remain.

We believe such a unified formulation is difficult, and we do not know how to do it.[7] Before inference rules, it's not even clear to us what the *judgments* should be in the logical formulation. However, the task does appear to have a vague precedent: Jensen [**Jen92**] expresses some forms of abstract interpretation in logical form, and Schmidt [**Sch12**] explores abstract interpretation from the perspectives of topology and denotational semantics. These works may be reasonable starting points.

We have only passing familiarity with the works of Jensen and Schmidt, and we do not know if their approaches can be adapted to our needs. To begin, we would need to study their approaches. We would have to identify the distinctions between our methods and their kinds of abstract interpretation (those kinds which they are reformulating). We would need to understand how these distinctions imply differences in the resulting logical formulation - if their approaches even apply in our case. Assuming we get this far, we would then need to study multiple forms of automatic deduction, to understand their strengths and weaknesses and how these fit with our logical formulation.

We think these questions are substantial and complex, and we leave them to future work.

5.3.7. Program verification

Throughout this work, we have focused on partial interpretation, as applied to partial evaluation. We end with a brief mention of a potential second application: automatic program verification.

Program verification deals in assertions, pre-conditions, post-conditions, and invariants. All of these can be seen as sets of values, within which actual values must stay confined. Conceptually, partial interpretation computes a set of possible values for any piece of data in a program. If the set of possible values is included in the asserted set, then that assertion has been verified.

Therefore, partial interpretation should have at least two customers: partial evaluation (for performance and generality), and program verification (for correctness). This makes its investigation doubly significant.

[7]We dismiss a mechanical translation of our algorithms to Prolog, which would factually reformulate our methods as inference rules, but achieve none of the benefits we just discussed.

References

[AIBB$^+$98] Norman I Adams IV, David H Bartley, Gary Brooks, R Kent Dybvig, Daniel Paul Friedman, Robert Halstead, Chris Hanson, Christopher Thomas Haynes, Eugene Kohlbecker, Don Oxley, et al. Revised5 report on the algorithmic language scheme. *ACM Sigplan Notices*, 33(9):26–76, 1998.

[And96] Peter Holst Andersen. Partial evaluation applied to ray tracing. In *Software Engineering im Scientific Computing*, pages 78–85. Springer, 1996.

[BAL07] Amir M Ben-Amram and Chin Soon Lee. Program termination analysis in polynomial time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1):1–37, 2007.

[BCD$^+$11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[BCD14] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. A coq formalization of the relational data model. In *European Symposium on Programming Languages and Systems*, pages 189–208. Springer, 2014.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[Ber89] Andrew A Berlin. A compilation strategy for numerical programs based on partial evaluation. 1989.

[BLR09] Carl Friedrich Bolz, Michael Leuschel, and Armin Rigo. Towards just-in-time partial evaluation of prolog. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 158–172. Springer, 2009.

[BST$^+$10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CGP$^+$08] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[CP95] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *ACM Sigplan Notices*, 30(3):35–49, 1995.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[Fee19] Marc Feeley. Gambit v4.9.3 manual. `http://www.iro.umontreal.ca/~gambit/doc/gambit.pdf`, February 2019.

[Flo67] Robert Floyd. Assigning meaning to programs. *Mathematical Aspects of Computer Science*, 19:19–31, 1967.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, 1991.

[Fut71] Yoshihiko Futamura. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, computers, controls*, 2(5):45–50, 1971.

[Glü09] Robert Glück. Is there a fourth futamura projection? In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 51–60, 2009.

[HKPM97] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The coq proof assistant a tutorial. *Rapport Technique*, 178, 1997.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[Jen92] Thomas Philip Jensen. Abstract interpretation in logical form. *PhD Thesis, the Imperial College*, 1992.

[JGS93] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[Kat98] Morris Joel Katz. *A new perspective on partial evaluation and use analysis*. stanford university, 1998.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[KW92] Morry Katz and Daniel Weise. Towards a new perspective on partial evaluation. *PEPM*, 92:19–20, 1992.

[KW94] Atsushi Kanamori and Daniel Weise. Worklist management strategies for dataflow analysis. Technical report, Technical Report MSR–TR–94–12, Microsoft Research, 1994.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[MM09] Matthew Might and Panagiotis Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 260–274. Springer, 2009.

[MNTHVH21] Cameron Moy, Phúc C Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Corpse reviver: sound and efficient gradual typing via contract verification. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

[Mos93] Christian Mossin. Partial evaluation of general parsers. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 13–21, 1993.

[Ngu19] Phúc Nguyen. *Higher-order Symbolic Execution.* PhD thesis, 2019.

[RKK17] Andrew Reynolds, Tim King, and Viktor Kuncak. Solving quantified linear arithmetic by counterexample-guided instantiation. *Formal Methods in System Design*, 51(3):500–532, 2017.

[RP06] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953, 2006.

[Ruf93] Erik Steven Ruf. *Topics in online partial evaluation.* PhD thesis, Citeseer, 1993.

[RWZ88] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 12–27, 1988.

[Sah91] Dan Sahlin. *An automatic partial evaluator for full Prolog.* Citeseer, 1991.

[Sch12] David A Schmidt. Inverse-limit and topological aspects of abstract interpretation. *Theoretical Computer Science*, 430:23–42, 2012.

[SMF21] Ian Sabourin, Stefan Monnier, and Marc Feeley. Computation over partial information. IFL'2021, `https://ifl21.cs.ru.nl/Program?action=download&upname=IFL21_Sabourin.pdf` or `http://www.iro.umontreal.ca/~monnier/ifl21-partial-interpretation.pdf`, August 2021.

[Tiw04] Ashish Tiwari. Termination of linear programs. In *International Conference on Computer Aided Verification*, pages 70–82. Springer, 2004.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, 2009.

[WCES94] Daniel Weise, Roger F Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 297–310, 1994.

[WCRS91] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 165–191. Springer, 1991.

[WWH$^+$17] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 662–676, 2017.

Appendix A

Consolidated algorithm from 3.3

We introduce the labels A1 to A4, for modifications made to satisfy our termination argument. Three parts of the argument led to modifications: P1 to P3 (see 3.4.4 to 3.4.6). The connections are as follows - as mentioned in 3.4.4, we ignore P1 in our implementation:

$$\text{P2} \longleftrightarrow \text{A1} \qquad\qquad \text{P3} \longleftrightarrow \text{A2, A3, A4}$$

Partial interpretation (top-level):

- get a cache entry for the root call, putting it on the worklist if it is fresh (3.3.2)
- flush the worklist, performing the worklist action (below) for each worklist item (3.3.5)
- read the result from the original cache entry

Worklist action:

- perform *pgraph* partial interpretation (below); if the result has changed:
  - [A1] if necessary, apply result widening (3.4.5) [$\rightarrow$ P2]
  - write to cache
  - put all our consumers on the worklist, and clear our consumer list (3.3.5)

pgraph partial interpretation – handle all node types:

- *result*: select an expression, reproducing `decls` and `props` (3.2.1)
- *const*: construct the right SMT expression (3.2.2)
- *entry* (3.3.2):
  - convert to extended comprehension (3.1.3)
  - mark the `decls` and `props` as originating from this node (3.2.3)
  - pipe to SMT (3.2.7)
- *prim*: combine the expressions, merging `decls` and `props` (3.2.2, 3.2.3)
- *prog* (3.2.4): lookup the program's index, and wrap it in a `mkProg`
- *exit*:
  - merge all the `decls` and `props` (3.2.3)
  - return the computed history for the results (3.2.4)

- *if* (3.2.7):
  - if the condition is $\bot$ (the empty set), then return $\bot$, with no `decls` or `props`
  - otherwise try to decide the condition
    - known true $\rightarrow$ return the result from the consequent branch
    - known false $\rightarrow$ return the result from the alternate branch
  - push `if` marker on the stack; get both branch results; pop `if` marker (3.4.6) [A2] [$\rightarrow$ P3]
  - then proceed as in (3.3.4):
    - both branches are $\bot \rightarrow$ return $\bot$
    - consequent branch is $\bot \rightarrow$ return the result from the alternate branch, and assert the negated condition
    - alternate branch is $\bot \rightarrow$ return the result from the consequent branch, and assert the condition
    - neither branch is $\bot \rightarrow$ return a conditional expression, merging `decls` and `props` (3.2.3)
- *call* (3.3 and 3.4):
  - check if any input is $\bot \rightarrow$ return $\bot$ (3.3.4)
  - ensure the called program is unique, otherwise return $\top$ (3.2.5)
  - merge all the `decls` and `props` (3.2.3)
  - convert to basic comprehension (3.1.3)
  - get a result from the cache:
    - lookup `(pgraph, args)` in the cache directly - upon a cache hit, select that result (3.3.2)
    - perform termination assessment, and possibly termination resolution (3.4.6) [A3] [$\rightarrow$ P3]
    - widen arguments if they have grown, and if they haven't been already by [A4] termination resolution (3.4.6) [$\rightarrow$ P3]
    - second lookup, creating a fresh cache entry if necessary
    - in any case, register the current analysis as a consumer of the cache entry we consume (3.3.5)
  - substitute from the formals to the arguments, in the result (3.3.1)
  - rename variables created by widening, to unique names in the caller's scope (3.4.5)
  - mark any new `decls` and `props` as originating from this node (3.2.3)
  - pipe to SMT (3.2.7)
  - additionally merge the `decls` and `props`, from the result and from the arguments (3.2.3)

Appendix B

Consolidated algorithm from 4.3

Modifications relative to Appendix A are shown in bold. We omit the connections with the termination argument. The only change relevant to termination is that we are now widening all arguments. In particular then, we are widening them when necessary for termination.

Partial interpretation (top-level):

- get a cache entry for the root call, putting it on the worklist if it is fresh (3.3.2)
- flush the worklist, performing the worklist action (below) for each worklist item (3.3.5)
- read the result from the original cache entry

Worklist action:

- perform *pgraph* partial interpretation (below)
- if the result has changed:
  - **if necessary, apply result widening (3.4.5, 4.3.5)**
  - write to cache
  - put all our consumers on the worklist, and clear our consumer list (3.3.5)

pgraph partial interpretation – handle all node types:

- *result*: select an expression, reproducing `decls` and `props` (3.2.1)
- *const*: construct the right SMT expression (3.2.2)
- *entry*:
  - **instantiate *any*s to fresh variables (4.3.4)**
  - mark the `decls` and `props` as originating from this node (3.2.3)
- *prim*:
  - **attempt to reduce syntactically known heads (4.3.4)**
  - otherwise, combine the expressions, merging `decls` and `props` (3.2.2, 3.2.3)
- *prog* (3.2.4): lookup the program's index, and wrap it in a `mkProg`
- *exit*:
  - merge all the `decls` and `props` (3.2.3)
  - return the computed history for the results (3.2.4)

- *if* (3.2.7):
  - if the condition is $\perp$ (the empty set), then return $\perp$, with no `decls` or `props`
  - otherwise try to decide the condition
    - known true $\rightarrow$ return the result from the consequent branch
    - known false $\rightarrow$ return the result from the alternate branch
  - push `if` marker on the stack; get both branch results; pop `if` marker (3.4.6)
  - then proceed as in (3.3.4):
    - both branches are $\perp$ $\rightarrow$ return $\perp$
    - consequent branch is $\perp$ $\rightarrow$ return the result from the alternate branch, and assert the negated condition
    - alternate branch is $\perp$ $\rightarrow$ return the result from the consequent branch, and assert the condition
    - neither branch is $\perp$ $\rightarrow$ return a conditional expression, **pulling out common heads (4.3.4)**, and merging `decls` and `props` (3.2.3)
- *call* (3.3):
  - check if any input is $\perp$ $\rightarrow$ return $\perp$ (3.3.4)
  - **in passive mode, check for an existing cache entry for this call (4.1.1)**
    - **if one exists, consume it as described below for active mode**
    - **otherwise, return** $\top$
  - in active mode, ensure the called program is unique, otherwise return $\top$ (3.2.5)
  - merge all the `decls` and `props` (3.2.3)
  - **widen the arguments (4.3.5)**
  - **if the arguments are unique, perform ordinary execution (4.3.2)**
  - get a result from the cache:
    - lookup `(pgraph, args)` in the cache directly - upon a cache hit, select that result (3.3.2)
    - perform termination assessment, and possibly termination resolution (3.4.6)
    - second lookup, creating a fresh cache entry if necessary
    - in any case, register the current analysis as a consumer of the cache entry we consume (3.3.5)
  - **once we have a result from the cache, consume it:**
    - **substitute from the unknown parts of the formals, to the corresponding parts of the arguments, in the result (4.3.4)**
    - rename variables created by widening, to unique names in the caller's scope (3.4.5)
    - mark any new `decls` and `props` as originating from this node (3.2.3)
    - additionally merge the `decls` and `props`, from the result and from the arguments (3.2.3)

Appendix C

Reference implementation

This PDF document is intended to be accompanied by our reference implementation. If you are missing our reference implementation, it should be available through the University of Montreal's archival service, Papyrus:

- go to `papyrus.bib.umontreal.ca`
- search for e.g. "Ian Sabourin computation over partial information"
- once you have found the right page, look for `Sabourin_Ian_2021_impl.zip`

It is not possible for us to provide a direct URL link there, from this PDF document. This is because the final PDF must be submitted, before that URL is created.

We provide our reference implementation for two purposes:

- so that our code may be examined
- so that the main experimental results we report may be reproduced.

Further instructions can be found with our reference implementation, in the text file `README`.

For any other purpose, we give no assurances.