THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Verified Self-Explaining Computation

Citation for published version:
Stolarek, J & Cheney, J 2019, Verified Self-Explaining Computation. in G Hutton (ed.), *Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019. Proceedings.* Lecture Notes in Computer Science, vol. 11825, Springer, Cham, pp. 76-102, 13th International Conference on Mathematics of Program Construction, Porto, Portugal, 7/10/19. https://doi.org/10.1007/978-3-030-33636-3_4

Digital Object Identifier (DOI):
10.1007/978-3-030-33636-3_4

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Mathematics of Program Construction

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 19. Apr. 2024

Verified Self-Explaining Computation

Jan Stolarek[1,2] and James Cheney[3,4]

[1] University of Edinburgh, UK jan.stolarek@ed.ac.uk
[2] Lodz University of Technology, Poland
[3] University of Edinburgh, UK jcheney@inf.ed.ac.uk
[4] The Alan Turing Institute

Abstract. Common programming tools, like compilers, debuggers, and IDEs, crucially rely on the ability to analyse program code to reason about its behaviour and properties. There has been a great deal of work on verifying compilers and static analyses, but far less on verifying dynamic analyses such as program slicing. Recently, a new mathematical framework for slicing was introduced in which forward and backward slicing are dual in the sense that they constitute a Galois connection. This paper formalizes forward and backward dynamic slicing algorithms for a simple imperative programming language, and formally verifies their duality using the Coq proof assistant.

1 Introduction

The aim of mathematical program construction is to proceed from (formal) specifications to (correct) implementations. For example, critical components such as compilers, and various static analyses they perform, have been investigated extensively in a formal setting [10]. However, we unfortunately do not yet live in a world where all programs are constructed in this way; indeed, since some aspects of programming (e.g. exploratory data analysis) appear resistant to *a priori* specification, one could debate whether such a world is even possible. In any case, today programs "in the wild" are not always mathematically constructed. What do we do then?

One answer is provided by a class of techniques aimed at *explanation*, *comprehension* or *debugging*, often based on run-time monitoring, and sometimes with a pragmatic or even ad hoc flavour. In our view, the mathematics of constructing well-founded (meta)programs for explanation are wanting [4]. For example, dynamic analyses such as *program slicing* have many applications in comprehending and restructuring programs, but their mathematical basis and construction are far less explored compared to compiler verification [2, 3].

Dynamic program slicing is a runtime analysis that identifies fragments of a program's input and source code – known together as a *program slice* – that were relevant to producing a chosen fragment of the output (a *slicing criterion*) [8, 21]. Slicing has a very large literature, and there are a wide variety of dynamic slicing algorithms. Most work on slicing has focused on imperative or object-oriented programs.

One common application of dynamic slicing is program debugging. Assume we have a program with variables `x`, `y`, and `z` and a programmer expects that after the program has finished running these variables will have respective values `1`, `2`, and `3`. If a programmer unfamiliar with the program finds that after execution, variable `y` contains `1` where she was expecting another value, she may designate `y` as a slicing criterion, and dynamic slicing will highlight fragments of the source code that could have contributed to producing the incorrect result. This narrows down the amount of code that the programmer needs to inspect to correct a program. In this tiny example, of course, there is not much to throw away and the programmer can just inspect the program — the real benefit of slicing is for understanding larger programs with multiple authors. Slicing can also be used for program comprehension, i.e. to understand the behaviour of an already existing program in order to re-engineer its specification, possibly non-existent or not up-to-date.

In recent years a new semantic basis for dynamic slicing has been proposed [15, 17]. It is based on the concept of Galois connections as known from order and lattice theory. Given lattices X and Y, a Galois connection is a pair of functions $g : Y \to X$ and $f : X \to Y$ such that $g(y) \leq x \iff y \leq f(x)$; then g is the *lower adjoint* and f is the *upper adjoint*. Galois connections have been advocated as a basis for mathematical program construction already, for example by Backhouse [1] and Mu and Oliveira [12]. They showed that if one can specify a problem space and show that it is one of the component functions of a Galois connection (the "easy" part), then *optimal solutions* to the problem (the "hard" part) are uniquely determined by the dual adjoint. A simple example arises from the duality between integer multiplication and division: the Galois connection $x \cdot y \leq z \iff x \leq z/y$ expresses that z/y is the greatest integer such that $(z/y) \cdot y \leq z$.

Whereas Galois connections have been used previously for constructing programs (as well as other applications such as program analysis), here we consider using Galois connections to construct programs for program slicing. In our setting, we consider lattices of *partial inputs* and *partial outputs* of a computation corresponding to possible input and output slicing criteria, as well as *partial programs* corresponding to possible slices—these are regarded as part of the input. We then define a forward semantics (corresponding to *forward slicing*) that expresses how much of the output of a program can be computed from a given partial input. Provided the forward semantics is monotone and preserves greatest lower bounds, it is the upper adjoint of a Galois connection, whose lower adjoint computes for each partial output the *smallest* partial input needed to compute it — which we consider an *explanation*. In other words, forward and backward slicing are *dual* in the sense that forward slicing computes "as much as possible" of the output given a partial input, while backward slicing computes "as little as needed" of the input to recover a partial output.

Figure 1 illustrates the idea for a small example where the "program" is an expression $(x + y, 2x)$, the input is an initial store containing $[x = 1, y = 2]$ and the output is the pair $(3, 2)$. Partial inputs, outputs, and programs are obtained

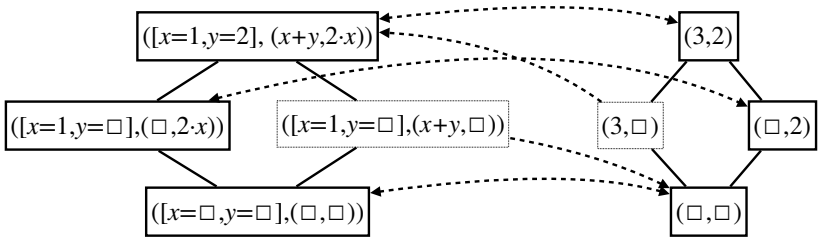

Fig. 1: Input and output lattices and Galois connection corresponding to expression $(x + y, 2 \cdot x)$ evaluated with input $[x = 1, y = 2]$ and output $(3, 2)$. Dotted lines with arrows pointing left calculate the lower adjoint, those pointing right calculate the upper adjoint, and lines with both arrows correspond to the induced isomorphism between the minimal inputs and maximal outputs. Several additional elements of the input lattice are omitted.

by replacing subexpressions by "holes" (□), as illustrated via (partial) lattice diagrams to the left and right. In the forward direction, computing the first component succeeds if both x and y are available while the second component succeeds if x is available; the backward direction computes the least input and program slice required for each partial output. Any Galois connection induces two isomorphic sublattices of the input and output, and in Figure 1 the elements of these sublattices are enclosed in boxes with thicker lines. In the input, these elements correspond to *minimal* explanations: partial inputs and slices in which every part is needed to explain the corresponding output. The corresponding outputs are *maximal* in a sense that their minimal explanations do not explain anything else in the output.

The Galois connection approach to slicing has been originally developed for functional programming languages [15] and then extended to functional languages with imperative features [17]. So far it has not been applied to conventional imperative languages, so it is hard to compare directly with conventional slicing techniques. Also, the properties of the Galois connection framework in [15, 17] have only been studied in pen-and-paper fashion. Such proofs are notoriously tricky in general and these are no exception; therefore, fully validating the metatheory of slicing based on Galois connections appears to be an open problem.

In this paper we present forward and backward slicing algorithms for a simple imperative language Imp and formally verify the correctness of these algorithms in Coq. Although Imp seems like a small and simple language, there are nontrivial technical challenges associated with slicing in the presence of mutable state, so our formalization provides strong assurance of the correctness of our solution. To the best of our knowledge, this paper presents the first formalization of a Galois connection slicing algorithm for an imperative language. Compared with Ricciotti et al. [17], Imp is a much simpler language than they consider, but

their results are not formalized; compared with Léchenet et al. [9], we formalize dynamic rather than static slicing.

In Section 2 we begin by reviewing the syntax of Imp, giving illustrative examples of slicing, and then reviewing the theory of slicing using the Galois connection framework, including the properties of minimality and consistency. In Section 3 we introduce an instrumented, tracing semantics for Imp and present the forward and backward slicing algorithms. We formalize all of the theory from Section 3 in Coq and prove their duality. Section 4 highlights key elements of our Coq development, with full code available online [19]. Section 5 provides pointers to other related work.

2 Overview

2.1 Imp slicing by example

Arithmetic expressions	$a ::= n \mid x \mid a_1 + a_2$
Boolean expressions	$b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid \neg b \mid b_1 \wedge b_2$
Imperative commands	$c ::= \texttt{skip} \mid x := a \mid c_1 \,;\, c_2$
	$\mid \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \mid \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \}$
Values	$v ::= v_a \mid v_b$
State	$\mu ::= \varnothing \mid \mu, x \mapsto v_a$ (x fresh)

Fig. 2: Imp syntax

For the purpose of our analysis we use a minimal imperative programming language Imp used in some textbooks on programming languages, e.g. [13, 16, 22][5]. Imp contains arithmetic and logical expressions, mutable state (a list of mappings from variable names to numeric values) and a small core of imperative commands: empty instruction (`skip`), variable assignments, instruction sequencing, conditional `if` instructions, and `while` loops. An Imp program is a series of commands combined using a sequencing operator. Imp lacks more advanced features, such as functions or pointers.

Figure 2 shows Imp syntax. We use letter n to denote natural number constants and x to denote program variables. In this presentation we have omitted subtraction ($-$) and multiplication ($\cdot$) from the list of arithmetic operators, and less-or-equal comparison ($\leq$) from the list of boolean operators. All these operators are present in our Coq development and we omit them here only to make the presentation in the paper more compact. Otherwise the treatment of subtraction and multiplication is analogous to the treatment of addition, and treatment of $\leq$ is analogous to $=$.

Dynamic slicing is a program simplification technique that determines which parts of the source code contributed to a particular program output. For example, a programmer might write this simple program in Imp:

[5] In the literature Imp is also referred to as WHILE.

```
if (y = 1) then { y := x + 1 }
           else { y := y + 1 } ;
z := z + 1
```

and after running it with an input state $[\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 0, \mathtt{z} \mapsto 2]$ might (wrongly) expect to obtain output state $[\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 3]$. However, after running the program the actual output state will be $[\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 3]$ with value of `y` differing from the expectation.

We can use dynamic slicing to debug this program by asking for an explanation which parts of the source code and the initial input state contributed to incorrect output value of `y`. We do this by formulating a *slicing criterion*, where we replace all values that we consider irrelevant in the output state (i.e. don't need an explanation for them) with *holes* (denoted with $\square$):

$$[\mathtt{x} \mapsto \square, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto \square]$$

and a slicing algorithm might produce a program slice:

```
if (y = 1) then { □ }
           else { y := y + 1 } ; □
```

with a sliced input state $[\mathtt{x} \mapsto \square, \mathtt{y} \mapsto 0, \mathtt{z} \mapsto \square]$. This result indicates which parts of the original source code and input state could be ignored when looking for a fault (indicated by replacing them with holes), and which ones were relevant in producing the output indicated in the slicing criterion. The result of slicing narrows down the amount of code a programmer has to inspect to locate a bug. Here we can see that only the input variable `y` was relevant in producing the result; `x` and `z` are replaced with a $\square$ in the input state, indicating their irrelevance. We can also see that the first branch of the conditional was not taken (unexpectedly!) and that in the second branch `y` was incremented to become 1. With irrelevant parts of the program hidden away it is now easier to spot that the problem comes from a mistake in the initial state. The initial value of `y` should be changed to 1 so that the first branch of the conditional is taken and then `y` obtains an output value of 2 as expected.

Consider the same example, but a different output slicing criterion $[\mathtt{x} \mapsto \square, \mathtt{y} \mapsto \square, \mathtt{z} \mapsto 3]$. In this case, a correctly sliced program is as follows:

```
□ ; z := z + 1
```

with the corresponding sliced input $[\mathtt{x} \mapsto \square, \mathtt{y} \mapsto \square, \mathtt{z} \mapsto 2]$, illustrating that the entire first conditional statement is irrelevant. However, while the conclusion seems intuitively obvious, actually calculating this correctly takes some work: we need to ensure that none of the assignments inside the taken conditional branch affected `z`, and conclude from this that the value of the conditional test `y = 1` is also irrelevant to the final value of `z`.

2.2 A Galois connection approach to program slicing

Example in Section 2.1 relies on an intuitive feel of how backward slicing should behave. We now address the question of how to make that intuition precise and show that slicing using the Galois connection framework introduced by Perera et al. [15] offers an answer.

Consider these two extreme cases of backward slicing behaviour:

1. for any slicing criterion backward slicing always returns a full program with no holes inserted;
2. for any slicing criterion backward slicing always returns a $\square$, i.e. it discards all the program code.

Neither of these two specifications is practically useful since they don't fulfil our intuitive expectation of "discarding program fragments irrelevant to producing a given fragment of program output". The first specification does not discard anything, bringing us no closer to understanding which code fragments are irrelevant. The second specification discards everything, including the code necessary to produce the output we want to understand. We thus want a backward slicing algorithm to have two properties:

- **consistency**: backward slicing retains code required to produce output we are interested in.
- **minimality**: backward slicing produces the smallest partial program and partial input state that suffice to achieve consistency.

Our first specification above does not have the minimality property; the second one does not have the consistency property. To achieve these properties we turn to order and lattice theory.

We begin with defining *partial Imp programs* (Figure 3) by extending Imp syntax presented in Figure 2 with *holes* (denoted using $\square$ in the semantic rules). A hole can appear in place of any arithmetic expression, boolean expression, or command. In the same way we allow values stored inside a state to be mapped to holes. For example:

$$\mu = [\mathtt{x} \mapsto 1, \mathtt{y} \mapsto \square]$$

is a *partial state* that maps variable `x` to 1 and variable `y` to a hole. We also introduce operation $\varnothing_\mu$ that takes a state μ and creates a partial state with the same domain as μ but all variables mapped to $\square$. For example if $\mu = [\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2]$ then $\varnothing_\mu = [\mathtt{x} \mapsto \square, \mathtt{y} \mapsto \square]$. A partial state that maps all its variables to holes is referred to as an *empty partial state*.

Having extended Imp syntax with holes, we define partial ordering relations on partial programs and partial states that consider holes to be syntactically smaller than any other subexpression. Figure 4 shows the partial ordering relation for arithmetic expressions. Definitions for ordering of partial boolean expressions and partial commands are analogous. Ordering for partial states is defined element-wise, thus requiring that two states in the ordering relation have identical domains, i.e. store the same variables in the same order.

Partial arithmetic expr.	$a ::= \ldots \mid \square$
Partial boolean expr.	$b ::= \ldots \mid \square$
Partial commands	$c ::= \ldots \mid \square$
Partial state	$\mu ::= \varnothing \mid \mu, x \mapsto v_a \mid \mu, x \mapsto \square$

Fig. 3: Partial Imp syntax. All elements of syntax from Figure 2 remain unchanged, only $\square$ are added.

$$\frac{}{\square \sqsubseteq a} \qquad \frac{}{n \sqsubseteq n} \qquad \frac{}{x \sqsubseteq x} \qquad \frac{a_1 \sqsubseteq a_1' \qquad a_2 \sqsubseteq a_2'}{a_1 + a_2 \sqsubseteq a_1' + a_2'}$$

Fig. 4: Ordering relation for partial arithmetic expressions.

For every Imp program p, a set of all partial programs smaller than p forms a complete finite lattice, written $\downarrow p$ with p being the top and $\square$ the bottom element of this lattice. Partial states, arithmetic expressions, and boolean expressions form lattices in the same way. Moreover, a pair of lattices forms a (product) lattice, with the ordering relation defined component-wise:

$$(a_1, b_1) \sqsubseteq (a_2, b_2) \iff a_1 \sqsubseteq a_2 \wedge b_1 \sqsubseteq b_2$$

Figure 5 shows definition of the *join* (*least upper bound*, $\sqcup$) operation for arithmetic expressions. Definitions for boolean expressions and imperative commands are analogous. A join exists for every two elements from a complete lattice formed by a program p or state μ [6, Theorem 2.31].

Assume we have a program p paired with an input state μ that evaluates to an output state μ'. We can now formulate slicing as a pair of functions between lattices:

- **forward slicing**: Forward slicing can be thought of as evaluation of partial programs. A function $\mathsf{fwd}_{(p,\mu)}$ takes as its input a partial program and a partial state from a lattice formed by pairing a program p and state μ. $\mathsf{fwd}_{(p,\mu)}$ outputs a partial state belonging to a lattice formed by μ'. The input to the forward slicing function is referred to as a *forward slicing criterion* and output as a *forward slice*.
- **backward slicing**: Backward slicing can be thought of as "rewinding" a program's execution. A function $\mathsf{bwd}_{\mu'}$ takes as its input a partial state from the lattice formed by the output state μ'. $\mathsf{bwd}_{\mu'}$ outputs a pair consisting of a partial program and a partial state, both belonging to a lattice formed by program p and state μ. Input to a backward slicing function is referred to as a *backward slicing criterion* and output as a *backward slice*.

A key point above (discussed in detail elsewhere [17]) is that for imperative programs, both $\mathsf{fwd}_{(p,\mu)}$ and $\mathsf{bwd}_{\mu'}$ depend not only on p, μ, μ' but also on the particular execution path taken while evaluating p on μ. (In earlier work on slicing pure functional programs [15], traces are helpful for *implementing* slicing

$$a \sqcup \square = a \qquad \square \sqcup a = a \qquad n \sqcup n = n \qquad x \sqcup x = x$$
$$(a_1 + a_2) \sqcup (a_1' + a_2') = (a_1 \sqcup a_1') + (a_2 \sqcup a_2')$$

Fig. 5: Join operation for arithmetic expressions.

efficiently but not required for *defining* it.) We make this information explicit in Section 3 by introducing *traces* T that capture the choices made during execution. We will define the slicing algorithms inductively as relations indexed by T, but in our Coq formalization $\mathsf{fwd}^T_{(p,\mu)}$ and $\mathsf{bwd}^T_{\mu'}$ are represented as dependently-typed functions where T is a proof term witnessing an operational derivation.

A pair of forward and backward slicing functions is guaranteed to have both the minimality and consistency properties when they form a Galois connection [6, Lemmas 7.26 and 7.33].

Definition 1 (Galois connection). *Given lattices P, Q and two functions $f : P \to Q$, $g : Q \to P$, we say f and g form a Galois connection (written $f \dashv g$) when $\forall_{p \in P, q \in Q} f(p) \sqsubseteq_Q q \iff p \sqsubseteq_P g(q)$. We call f a* lower adjoint *and g an* upper adjoint.

Importantly, for a given Galois connection $f \dashv g$, function f uniquely determines g and vice versa [6, Lemma 7.33]. This means that our choice of fwd (i.e. definition of how to evaluate partial programs on partial inputs) uniquely determines the backward slicing function bwd that will be minimal and consistent with respect to fwd, provided we can show that fwd and bwd form a Galois connection. There are many strategies to show that two functions $f : P \to Q$ and $g : Q \to P$ form a Galois connection, or to show that f or g in isolation has an upper or respectively lower adjoint. One attractive approach is to show that f preserves least upper bounds, or dually that g preserves greatest lower bounds (in either case, monotonicity follows as well). This approach is indeed attractive because it allows us to analyse just one of f or g and know that its dual adjoint exists, without even having to write it down. Indeed, in previous studies of Galois slicing [15, 17], this characterization was the one used: fwd was shown to preserve greatest lower bounds to establish the existence of its lower adjoint bwd, and then efficient versions of bwd were defined and proved correct.

For our constructive formalization, however, we really want to give computable definitions for both fwd and bwd and prove they form a Galois connection, so while preservation of greatest lower bounds by fwd is a useful design constraint, proving it does not really save us any work. Instead, we will use the following equivalent characterization of Galois connections [6, Lemma 7.26]:

1. f and g are monotone
2. *deflation* property holds:

$$\forall_{q \in Q} \; f(g(q)) \sqsubseteq_Q q$$

$$
\begin{array}{lll}
\text{Arithmetic traces} & T_a & ::= n \mid x(v_a) \mid T_{a1} + T_{a2} \\
\text{Boolean traces} & T_b & ::= \texttt{true} \mid \texttt{false} \mid T_{a1} = T_{a2} \mid \neg T_b \mid T_{b1} \wedge T_{b2} \\
\text{Command traces} & T_c & ::= \texttt{skip} \mid x := T_a \mid T_1 \,;\, T_2 \\
 & & \mid\ \texttt{if}_{\textsf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} \mid \texttt{if}_{\textsf{false}}\ T_b\ \texttt{else}\ \{\ T_2\ \} \\
 & & \mid\ \texttt{while}_{\textsf{false}}\ T_b \mid \texttt{while}_{\textsf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w
\end{array}
$$

Fig. 6: Trace syntax

3. *inflation* property holds:

$$\forall_{p \in P}\ \ p \sqsubseteq_P g(f(p))$$

We use this approach in our Coq mechanization. We will first prove a general theorem that any pair of functions that fulfils properties (1)–(3) above forms a Galois connection. We will then define forward and backward slicing functions for Imp programs and prove that they are monotone, deflationary, and inflationary. Once this is done we will instantiate the general theorem with our specific definitions of forward and backward slicing to arrive at the proof that our slicing functions form a Galois connection. This is the crucial correctness property that we aim to prove. We also prove that existence of a Galois connection between forward and backward slicing functions implies consistency and minimality properties. Note that consistency is equivalent to the inflation property.

3 Dynamic program slicing

3.1 Tracing semantics

Following previous work [15, 17], we employ a *tracing semantics* to define the slicing algorithms. Since dynamic slicing takes account of the actual execution path followed by a run of a program, we represent the execution path taken using an explicit trace data structure. Traces are then traversed as part of both the forward and backward slicing algorithms. That is, unlike tracing evaluation, forward and backward slicing follow the structure of traces, rather than the program. Note that we are not really inventing anything new here: in our formalization, the trace is simply a *proof term* witnessing the derivability of the operational semantics judgement. The syntax of traces is shown in Figure 6. The structure of traces follows the structure of language syntax with the following exceptions:

- the expression trace $x(v_a)$ records both the variable name x and a value v_a that was read from program state μ;
- for conditional instructions, traces record which branch was actually taken. When the `if` condition evaluates to **true** we store traces of evaluating the condition and the `then` branch; if it evaluates to **false** we store traces of evaluating the condition and the `else` branch.

– for `while` loops, if the condition evaluates to **false** (i.e. loop body does not execute) we record only a trace for the condition. If the condition evaluates to **true** we record traces for the condition (T_b), a single execution of the loop body (T_c) and the remaining iterations of the loop (T_w).

$$\frac{}{\mu, n \Rightarrow n :: v_n} \qquad \frac{\mu(x) = v_a}{\mu, x \Rightarrow x(v_a) :: v_a} \qquad \frac{\mu, a_1 \Rightarrow T_1 :: v_1 \qquad \mu, a_2 \Rightarrow T_2 :: v_2}{\mu, a_1 + a_2 \Rightarrow T_1 + T_2 :: v_1 +_{\mathbb{N}} v_2}$$

Fig. 7: Imp arithmetic expressions evaluation

$$\frac{}{\mu, \texttt{true} \Rightarrow \texttt{true} :: \textbf{true}} \qquad \frac{}{\mu, \texttt{false} \Rightarrow \texttt{false} :: \textbf{false}}$$

$$\frac{\mu, a_1 \Rightarrow T_1 :: v_1 \qquad \mu, a_2 \Rightarrow T_2 :: v_2}{\mu, a_1 = a_2 \Rightarrow T_1 = T_2 :: v_1 =_{\mathbb{B}} v_2} \qquad \frac{\mu, b \Rightarrow T :: v}{\mu, \neg b \Rightarrow \neg T :: \neg_{\mathbb{B}} v}$$

$$\frac{\mu, b_1 \Rightarrow T_1 :: v_1 \qquad \mu, b_2 \Rightarrow T_2 :: v_2}{\mu, b_1 \wedge b_2 \Rightarrow T_1 \wedge T_2 :: v_1 \wedge_{\mathbb{B}} v_2}$$

Fig. 8: Imp boolean expressions evaluation

Figures 7–9 show evaluation rules for arithmetic expressions, boolean expressions, and imperative commands, respectively[6]. Traces are written in grey colour and separated with a double colon (::) from the evaluation result. Arithmetic expressions evaluate to numbers (denoted v_a). Boolean expressions evaluate to either **true** or **false** (jointly denoted as v_b). Operators with $_{\mathbb{N}}$ or $_{\mathbb{B}}$ subscripts should be evaluated as mathematical and logical operators respectively to arrive at an actual value; this is to distinguish them from the language syntax. Commands evaluate by side-effecting on the input state, producing a new state as output (Figure 9). Only arithmetic values can be assigned to variables and stored inside a state. Assignments to variables absent from the program state are treated as no-ops. This means all variables that we want to write and read must be included (initialized) in the initial program state. We explain reasons behind this decision later in Section 4.3.

3.2 Forward slicing

In this and the next section we present concrete definitions of forward and backward slicing for Imp programs. Readers may prefer to skip ahead to Section 3.4 for an extended example of these systems at work first. Our slicing algorithms are based on the ideas first presented in [17]. Presentation in Section 2.2 views

[6] We overload the $\Rightarrow$ notation to mean one of three evaluation relations. It is always clear from the arguments which relation we are referring to.

$$\frac{}{\mu, \texttt{skip} \Rightarrow \texttt{skip} :: \mu} \qquad \frac{\mu, a \Rightarrow_a T_a :: v_a}{\mu, x := a \Rightarrow x := T_a :: \mu[x \mapsto v_a]}$$

$$\frac{\mu, c_1 \Rightarrow T_1 :: \mu' \qquad \mu', c_2 \Rightarrow T_2 :: \mu''}{\mu, c_1 \,;\, c_2 \Rightarrow T_1 \,;\, T_2 :: \mu''}$$

$$\frac{\mu, b \Rightarrow T_b :: \textsf{true} \qquad \mu, c_1 \Rightarrow T_1 :: \mu'}{\mu, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \Rightarrow \texttt{if}_{\textsf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu'}$$

$$\frac{\mu, b \Rightarrow T_b :: \textsf{false} \qquad \mu, c_2 \Rightarrow T_2 :: \mu'}{\mu, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \Rightarrow \texttt{if}_{\textsf{false}}\ T_b\ \texttt{else}\ \{\ T_2\ \} :: \mu'}$$

$$\frac{\mu, b \Rightarrow T_b :: \textsf{false}}{\mu, \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \Rightarrow \texttt{while}_{\textsf{false}}\ T_b :: \mu}$$

$$\frac{\mu, b \Rightarrow T_b :: \textsf{true} \qquad \mu, c \Rightarrow T_c :: \mu' \qquad \mu', \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \Rightarrow T_w :: \mu''}{\mu, \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \Rightarrow \texttt{while}_{\textsf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu''}$$

Fig. 9: Imp command evaluation.

$$\frac{}{T :: \mu, \square \nearrow \square} \qquad \frac{}{n :: \mu, n \nearrow n} \qquad \frac{}{x(v_a) :: \mu, x \nearrow \mu(x)}$$

$$\frac{T_1 :: \mu, a_1 \nearrow \square}{T_1 + T_2 :: \mu, a_1 + a_2 \nearrow \square} \qquad \frac{T_2 :: \mu, a_2 \nearrow \square}{T_1 + T_2 :: \mu, a_1 + a_2 \nearrow \square}$$

$$\frac{T_1 :: \mu, a_1 \nearrow v_1 \qquad T_2 :: \mu, a_2 \nearrow v_2}{T_1 + T_2 :: \mu, a_1 + a_2 \nearrow v_1 +_{\mathbb{N}} v_2} \quad v_1, v_2 \neq \square$$

Fig. 10: Forward slicing rules for Imp arithmetic expressions.

the slicing algorithms as computable functions and we will implement them in code as such. However for the purpose of writing down the formal definitions of our algorithms we will use a relational notation. It is more concise and allows easier comparisons with previous work.

Figures 10–12 present forward slicing rules for the Imp language[7]. As mentioned in Section 2.2, forward slicing can be thought of as evaluation of partial programs. Thus the forward slicing relations $\nearrow$ take a partial program, a partial state, and an execution trace as an input and return a partial value, either a partial number (for partial arithmetic expressions), a partial boolean (for partial logical expressions) or a partial state (for partial commands). For example, we can read $T :: \mu, c \nearrow \mu'$ as "Given trace T, in partial environment μ the partial command c forward slices to partial output μ'."

A general principle in the forward slicing rules for arithmetic expressions (Figure 10) and logical expressions (Figure 11) is that "holes propagate". This

[7] We again overload $\nearrow$ and $\searrow$ arrows in the notation to denote one of three forward/backward slicing relations. This is important in the rules for boolean slicing, whose premises refer to the slicing relation for arithmetic expressions, and command slicing, whose premises refer to slicing relation for boolean expressions.

$$\frac{}{T :: \mu, \square \nearrow \square}$$

$$\frac{}{\texttt{true} :: \mu, \texttt{true} \nearrow \textbf{true}} \qquad \frac{}{\texttt{false} :: \mu, \texttt{false} \nearrow \textbf{false}}$$

$$\frac{T_1 :: a_1 \nearrow \square}{T_1 = T_2 :: \mu, a_1 = a_2 \nearrow \square} \qquad \frac{T_2 :: a_2 \nearrow \square}{T_1 = T_2 :: \mu, a_1 = a_2 \nearrow \square}$$

$$\frac{T_1 :: a_1 \nearrow v_1 \qquad T_2 :: a_2 \nearrow v_2}{T_1 = T_2 :: \mu, a_1 = a_2 \nearrow v_1 =_{\mathbb{B}} v_2} \quad v_1, v_2 \neq \square$$

$$\frac{T :: b \nearrow \square}{\neg T :: \mu, \neg b \nearrow \square} \qquad \frac{T :: b \nearrow v_b}{\neg T :: \mu, \neg b \nearrow \neg_{\mathbb{B}} v} \quad v_b \neq \square$$

$$\frac{T_1 :: b_1 \nearrow \square}{T_1 \wedge T_2 :: \mu, b_1 \wedge b_2 \nearrow \square} \qquad \frac{T_2 :: b_2 \nearrow \square}{T_1 \wedge T_2 :: \mu, b_1 \wedge b_2 \nearrow \square}$$

$$\frac{T_1 :: b_1 \nearrow v_1 \qquad T_2 :: b_2 \nearrow v_2}{T_1 \wedge T_2 :: \mu, b_1 \wedge b_2 \nearrow v_1 \wedge_{\mathbb{B}} v_2} \quad v_1, v_2 \neq \square$$

Fig. 11: Forward slicing rules for Imp boolean expressions.

means that whenever $\square$ appears as an argument of an operator, application of that operator forward slices to a $\square$. For example, $1 + \square$ forward slices to a $\square$ and so does $\neg\square$. In other words, if an arithmetic or logical expression contains at least one hole it will reduce to a $\square$; if it contains no holes it will reduce to a proper value. This is not the case for commands though. For example, command `if true then` 1 `else` $\square$ forward slices to 1, even though it contains a hole in the (not taken) `else` branch.

A rule worth attention is one for forward slicing of variable reads:

$$\frac{}{x(v_a) :: \mu, x \nearrow \mu(x)}$$

It is important here that we read the return value from μ and not v_a recorded in a trace. This is because μ is a partial state and also part of a forward slicing criterion. It might be that μ maps x to $\square$, in which case we must forward slice to $\square$ and not to v_a. Otherwise minimality would not hold.

Forward slicing rules for arithmetic and logical expressions both have a universal rule for forward slicing of holes that applies regardless of what the exact trace value is:

$$\frac{}{T :: \mu, \square \nearrow \square}$$

There is no such rule for forward slicing of commands (Figure 12). There we have separate rules for forward slicing of holes for each possible trace. This is due to the side-effecting nature of commands, which can mutate the state through variable assignments. Consider this rule for forward slicing of assignments w.r.t.

$$\frac{}{\texttt{skip} :: \mu, \square \nearrow \mu} \qquad \frac{}{\texttt{skip} :: \mu, \texttt{skip} \nearrow \mu} \qquad \frac{}{x := T_a :: \mu, \square \nearrow \mu[x \mapsto \square]}$$

$$\frac{T_a :: \mu, a \nearrow v_a}{x := T_a :: \mu, x := a \nearrow \mu[x \mapsto v_a]} \qquad \frac{T_1 :: \mu, \square \nearrow \mu' \qquad T_2 :: \mu', \square \nearrow \mu''}{T_1 \,;\, T_2 :: \mu, \square \nearrow \mu''}$$

$$\frac{T_1 :: \mu, c_1 \nearrow \mu' \qquad T_2 :: \mu', c_2 \nearrow \mu''}{T_1 \,;\, T_2 :: \mu, c_1 \,;\, c_2 \nearrow \mu''} \qquad \frac{T_1 :: \mu, \square \nearrow \mu'}{\texttt{if}_{\textsf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu, \square \nearrow \mu'}$$

$$\frac{T_b :: \mu, b \nearrow \square \qquad T_1 :: \mu, \square \nearrow \mu'}{\texttt{if}_{\textsf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \nearrow \mu'}$$

$$\frac{T_b :: \mu, b \nearrow v_b \qquad T_1 :: \mu, c_1 \nearrow \mu'}{\texttt{if}_{\textsf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \nearrow \mu'} \quad v_b \neq \square$$

$$\frac{T_2 :: \mu, \square \nearrow \mu'}{\texttt{if}_{\textsf{false}}\ T_b\ \texttt{else}\ \{\ T_2\ \} :: \mu, \square \nearrow \mu'}$$

$$\frac{T_b :: \mu, b \nearrow \square \qquad T_2 :: \mu, \square \nearrow \mu'}{\texttt{if}_{\textsf{false}}\ T_b\ \texttt{else}\ \{\ T_2\ \} :: \mu, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \nearrow \mu'}$$

$$\frac{T_b :: \mu, b \nearrow v_b \qquad T_2 :: \mu, c_2 \nearrow \mu'}{\texttt{if}_{\textsf{false}}\ T_b\ \texttt{else}\ \{\ T_2\ \} :: \mu, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ c_2\ \} \nearrow \mu'} \quad v_b \neq \square$$

$$\frac{}{\texttt{while}_{\textsf{false}}\ T_b :: \mu, \square \nearrow \mu} \qquad \frac{}{\texttt{while}_{\textsf{false}}\ T_b :: \mu, \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \nearrow \mu}$$

$$\frac{T_c :: \mu, \square \nearrow \mu_c \qquad T_w :: \mu_c, \square \nearrow \mu_w}{\texttt{while}_{\textsf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu, \square \nearrow \mu_w}$$

$$\frac{T_b :: \mu, b \nearrow \square \qquad T_c :: \mu, \square \nearrow \mu_c \qquad T_w :: \mu_c, \square \nearrow \mu_w}{\texttt{while}_{\textsf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu, \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \nearrow \mu_w}$$

$$\frac{T_b :: \mu, b \nearrow v_b \qquad T_c :: \mu, c \nearrow \mu_c \qquad T_w :: \mu_c, \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \nearrow \mu_w}{\texttt{while}_{\textsf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu, \texttt{while}\ b\ \texttt{do}\ \{\ c\ \} \nearrow \mu_w} \quad v_b \neq \square$$

Fig. 12: Forward slicing rules for Imp commands.

a $\square$ as a slicing criterion[8]:

$$\frac{}{x := T_a :: \mu, \square \nearrow \mu[x \mapsto \square]}$$

When forward slicing an assignment w.r.t. a $\square$ we need to erase (i.e. change to a $\square$) variable x in the state μ, which follows the principle of "propagating holes". Here having a trace is crucial to know which variable was actually assigned during the execution. Rules for forward slicing of other commands w.r.t. a $\square$ traverse the trace recursively to make sure that all variable assignments within a trace are reached. For example:

$$\frac{T_1 :: \mu, \square \nearrow \mu'}{\texttt{if}_{\textsf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu, \square \nearrow \mu'}$$

[8] When some partial value v is used as a slicing criterion we say that we "slice w.r.t. v".

In this rule, trace T_1 is traversed recursively to arrive at a state μ' that is then returned as the final product of the rule. Notice how trace T_b is *not* traversed. This is because boolean expressions (and arithmetic ones as well) do not have side effects on the state and so there is no need to traverse them.

The problem of traversing the trace recursively to handle side-effects to the state can be approached differently. Authors of [17] have formulated a single rule, which we could adapt to our setting like this:

$$\frac{\mathcal{L} = \mathsf{writes}(T)}{T :: \mu, \Box \nearrow \mu \triangleleft \mathcal{L}}$$

In this rule $\mathsf{writes}(T)$ means "all state locations written to inside trace T" and $\mu \triangleleft \mathcal{L}$ means erasing (i.e. mapping to a $\Box$) all locations in μ that are mentioned in $\mathcal{L}$. Semantically this is equivalent to our rules – both approaches achieve the same effect. However, we have found having separate rules easier to formalize in a proof assistant.

3.3 Backward slicing

Backward slicing rules are given in Figures 13–15. These judgements should be read left-to-right, for example, $T :: \mu \searrow \mu', c$ should be read as "Given trace T and partial output state μ, backward slicing yields partial input μ' and partial command c." Their intent is to reconstruct the smallest program code and initial state that suffice to produce, after forward slicing, a result that is at least as large as the backward slicing criterion. To this end, backward slicing crucially relies on execution traces as part of input, since slicing effectively runs a program backwards (from final result to source code).

Figures 13 and 14 share a universal rule for backward slicing w.r.t. a hole.

$$\frac{}{T :: \mu, \Box \searrow \varnothing_\mu, \Box}$$

This rule means that to obtain an empty result it always suffices to have an empty state and no program code. This rule always applies preferentially over other rules, which means that whenever a value, such as v_a or v_b, appears as a backward slicing criterion we know it is not a $\Box$. Similarly, Figure 15 has a rule:

$$\frac{}{T :: \varnothing \searrow \varnothing_\varnothing, \Box}$$

It means that backward slicing w.r.t. a state with an empty domain (i.e. containing no variables) returns an empty partial state and an empty program. Of course having a program operating over a state with no variables would be completely useless – since a state cannot be extended with new variables during execution we wouldn't observe any effects of such a program. However, in the Coq formalization, it is necessary to handle this case because otherwise Coq will not accept that the definition of backward slicing is a total function.

In the rule for backward slicing of variable reads (third rule in Figure 13) it might seem that v_a stored inside a trace is redundant because we know what v_a

$$\frac{}{T :: \mu, \square \searrow \varnothing_\mu, \square} \qquad \frac{}{n :: \mu, v_a \searrow \varnothing_\mu, n}$$

$$\frac{}{x(v_a) :: \mu, v_a \searrow \varnothing_\mu[x \mapsto v_a], x}$$

$$\frac{T_1 :: \mu, v_1 \searrow \mu_1, a_1 \qquad T_2 :: \mu, v_2 \searrow \mu_2, a_2}{T_1 + T_2 :: \mu, v_a \searrow \mu_1 \sqcup \mu_2, a_1 + a_2}$$

Fig. 13: Backward slicing rules for Imp arithmetic expressions.

$$\frac{}{T :: \mu, \square \searrow \varnothing_\mu, \square}$$

$$\frac{}{\texttt{true} :: \mu, \textbf{true} \searrow \varnothing_\mu, \texttt{true}} \qquad \frac{}{\texttt{false} :: \mu, \textbf{false} \searrow \varnothing_\mu, \texttt{false}}$$

$$\frac{T_1 :: \mu, v_1 \searrow \mu_1, a_1 \qquad T_2 :: \mu, v_2 \searrow \mu_2, a_2}{T_1 = T_2 :: \mu, v_b \searrow \mu_1 \sqcup \mu_2, a_1 = a_2}$$

$$\frac{T :: \mu, v_b \searrow \mu', b}{\neg T :: \mu, v_b \searrow \mu', \neg b}$$

$$\frac{T_1 :: \mu, v_1 \searrow \mu_1, b_1 \qquad T_2 :: \mu, v_2 \searrow \mu_2, b_2}{T_1 \wedge T_2 :: \mu, v_b \searrow \mu_1 \sqcup \mu_2, b_1 \wedge b_2}$$

Fig. 14: Backward slicing rules for Imp boolean expressions.

is from the slicing criterion. This is a way of showing that variables can only be sliced w.r.t. values they have evaluated to during execution. So for example if x evaluated to 17 it is not valid to backward slice it w.r.t. 42.

The rule for backward slicing of addition in Figure 13 might be a bit surprising. Each of the subexpressions is sliced w.r.t. a value that this expression has evaluated to (v_1, v_2), and not w.r.t. v_a. It might seem we are getting v_1 and v_2 out of thin air, since they are not directly recorded in a trace. Note however that knowing T_1 and T_2 allows to recover v_1 and v_2 at the expense of additional computations. In the actual implementation we perform induction on the structure of evaluation derivations, which record values of v_1 and v_2, thus allowing us to avoid extra computations. We show v_1 and v_2 in our rules but avoid showing the evaluation relation as part of slicing notation. This is elaborated further in Section 4.4.

Recursive backward slicing rules also rely crucially on the join ($\sqcup$) operation, which combines smaller slices from slicing subexpressions into one slice for the whole expression.

There are two separate rules for backward slicing of variable assignments (rules 3 and 4 in Figure 15). If a variable is mapped to a $\square$ it means it is irrelevant. We therefore maintain mapping to a $\square$ and do not reconstruct variable assignment instructions. If a variable is relevant though, i.e. it is mapped to a concrete value in a slicing criterion, we reconstruct the assignment instruction

$$\frac{}{T :: \varnothing \searrow \varnothing_{\varnothing}, \square} \qquad \frac{}{\texttt{skip} :: \mu \searrow \mu, \square} \qquad \frac{}{x := T_a :: \mu[x \mapsto \square] \searrow \mu[x \mapsto \square], \square}$$

$$\frac{T_a :: v_a \searrow \mu_a, a}{x := T_a :: \mu[x \mapsto v_a] \searrow \mu_a \sqcup \mu[x \mapsto \square], x := a} \quad v_a \neq \square$$

$$\frac{T_2 :: \mu \searrow \mu', \square \qquad T_1 :: \mu' \searrow \mu'', \square}{T_1 \,;\, T_2 :: \mu \searrow \mu'', \square} \qquad \frac{T_2 :: \mu \searrow \mu', \square \qquad T_1 :: \mu' \searrow \mu'', c_1}{T_1 \,;\, T_2 :: \mu \searrow \mu'', c_1 \,;\, \square} \quad c_1 \neq \square$$

$$\frac{T_2 :: \mu \searrow \mu', c_2 \qquad T_1 :: \mu' \searrow \mu'', c_1}{T_1 \,;\, T_2 :: \mu \searrow \mu'', c_1 \,;\, c_2} \quad c_2 \neq \square \qquad \frac{T_1 :: \mu \searrow \mu', \square}{\texttt{if}_{\textsf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu \searrow \mu', \square}$$

$$\frac{T_1 :: \mu \searrow \mu', c_1 \qquad T_b :: \texttt{true} \searrow \mu_b, b}{\texttt{if}_{\textsf{true}}\ T_b\ \texttt{then}\ \{\ T_1\ \} :: \mu \searrow \mu' \sqcup \mu_b, \texttt{if}\ b\ \texttt{then}\ \{\ c_1\ \}\ \texttt{else}\ \{\ \square\ \}} \quad c_1 \neq \square$$

$$\frac{T_2 :: \mu \searrow \mu', \square}{\texttt{if}_{\textsf{false}}\ T_b\ \texttt{else}\ \{\ T_2\ \} :: \mu \searrow \mu', \square}$$

$$\frac{T_2 :: \mu \searrow \mu', c_2 \qquad T_b :: \texttt{false} \searrow \mu_b, b}{\texttt{if}_{\textsf{false}}\ T_b\ \texttt{else}\ \{\ T_2\ \} :: \mu \searrow \mu' \sqcup \mu_b, \texttt{if}\ b\ \texttt{then}\ \{\ \square\ \}\ \texttt{else}\ \{\ c_2\ \}} \quad c_2 \neq \square$$

$$\frac{}{\texttt{while}_{\textsf{false}}\ T_b :: \mu \searrow \mu, \square} \qquad \frac{T_w :: \mu \searrow \mu_w, \square \qquad T_c :: \mu_w \searrow \mu_c, \square}{\texttt{while}_{\textsf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu \searrow \mu_c, \square}$$

$$\frac{T_w :: \mu \searrow \mu_w, \square \qquad T_c :: \mu_w \searrow \mu_c, c \qquad T_b :: \texttt{true} \searrow \mu_b, b}{\texttt{while}_{\textsf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu \searrow \mu_c \sqcup \mu_b, \texttt{while}\ b\ \texttt{do}\ \{\ c\ \}} \quad c \neq \square$$

$$\frac{T_w :: \mu \searrow \mu_w, c_w \qquad T_c :: \mu_w \searrow \mu_c, c \qquad T_b :: \texttt{true} \searrow \mu_b, b}{\texttt{while}_{\textsf{true}}\ T_b\ \texttt{do}\ \{\ T_c\ \};\ T_w :: \mu \searrow \mu_c \sqcup \mu_b, c_w \sqcup \texttt{while}\ b\ \texttt{do}\ \{\ c\ \}} \quad c_w \neq \square$$

Fig. 15: Backward slicing rules for Imp commands.

together with an arithmetic expression in the RHS. We also join state μ_a required to evaluate the RHS with $\mu[\texttt{x} \mapsto \square]$. It is crucial that we erase x in μ prior to joining. Firstly, if `x` is assigned, its value becomes irrelevant prior to the assignment, unless `x` is read during evaluation of the RHS (e.g. we are slicing an assignment `x := x + 1`). In this case `x` will be included in μ_a but its value can be different than the one in μ. It is thus necessary to erase `x` in μ to make a join operation possible.

At this point, it may be helpful to review the forward rules for assignment and compare with the backward rules, illustrated via a small example. Suppose we have an assignment `z := x + y`, initially evaluated on $[\texttt{w} \mapsto 0, \texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 42]$, and yielding result state $[\texttt{w} \mapsto 0, \texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 3]$. The induced

lattice of minimal inputs and maximal outputs consists of the following pairs:

$$[\mathtt{w} \mapsto v, \mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto \square] \longleftrightarrow [\mathtt{w} \mapsto v, \mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 3]$$
$$[\mathtt{w} \mapsto v, \mathtt{x} \mapsto 1, \mathtt{y} \mapsto \square, \mathtt{z} \mapsto \square] \longleftrightarrow [\mathtt{w} \mapsto v, \mathtt{x} \mapsto 1, \mathtt{y} \mapsto \square, \mathtt{z} \mapsto \square]$$
$$[\mathtt{w} \mapsto v, \mathtt{x} \mapsto \square, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto \square] \longleftrightarrow [\mathtt{w} \mapsto v, \mathtt{x} \mapsto \square, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto \square]$$
$$[\mathtt{w} \mapsto v, \mathtt{x} \mapsto \square, \mathtt{y} \mapsto \square, \mathtt{z} \mapsto \square] \longleftrightarrow [\mathtt{w} \mapsto v, \mathtt{x} \mapsto \square, \mathtt{y} \mapsto \square, \mathtt{z} \mapsto \square]$$

where $\mathtt{v} \in \{\square, 0\}$ so that each line above abbreviates two concrete relationships; the lattice has the shape of a cube. Because w is not read or written by `z := x + y`, it is preserved if present in the forward direction or if required in the backward direction. Because `z` is written but not read, its initial value is always irrelevant. To obtain the backward slice of any other partial output, such as $[\mathtt{w} \mapsto \square, \mathtt{x} \mapsto 1, \mathtt{y} \mapsto \square, \mathtt{z} \mapsto 3]$, find the smallest maximal partial output containing it, and take its backward slice, e.g. $[\mathtt{w} \mapsto \square, \mathtt{x} \mapsto 1, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto \square]$.

In the backward slicing rules for `if` instructions, we only reconstruct a branch of the conditional that was actually taken during execution, leaving a second branch as a $\square$. Importantly in these rules state μ_b is a minimal state sufficient for an `if` condition to evaluate to a `true` or `false` value. That state is joined with state μ', which is a state sufficient to evaluate the reconstructed branch of an `if`.

Rules for `while` slicing follow a similar approach. It might seem that the second rule for slicing $\mathtt{while}_{\mathsf{true}}$ is redundant because it is a special case of the third $\mathtt{while}_{\mathsf{true}}$ rule if we allowed $c_w = \square$. Indeed, that is the case on paper. However, for the purpose of a mechanized formalization we require that these two rules are separate. This shows that formalizing systems designed on paper can indeed be tricky and require modifications tailored to solve mechanization-specific issues.

Readers might have noticed that whenever a backward slicing rule from Figure 15 returns $\square$ as an output program, the state returned by the rule will be identical to the input state. One could then argue that we should reflect this in our rules by explicitly denoting that input and output states are the same, e.g.

$$\frac{T_1 :: \mu \searrow \mu, \square}{\mathtt{if}_{\mathsf{true}}\ T_b\ \mathtt{then}\ \{\ T_1\ \} :: \mu \searrow \mu, \square}$$

While it is true that in such a case states will be equal, this approach would not be directly reflected in the implementation, where slicing is implemented as a function and a result is always assigned to a new variable. However, it would be possible to prove a lemma about equality of input and output states for $\square$ output programs, should we need this fact.

3.4 An Extended Example of Backward Slicing

We now turn to an extended example that combines all the programming constructs of Imp[9]: assignments, sequencing, conditionals, and loops. Figure 16

[9] This example is adapted from [9].

shows a program that divides integer `a` by `b`, and produces a quotient `q`, remainder `r`, and result `res` that is set to 1 if `b` divides `a` and to 0 otherwise.

$[\mathtt{q} \mapsto 0, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 0, \mathtt{a} \mapsto 4, \mathtt{b} \mapsto 2]$

```
r := a;
while ( b <= r ) do {
  q := q + 1;
  r := r - b
};
if ( ! (r = 0) )
then { res := 0 }
else { res := 1 }
```

$[\mathtt{q} \mapsto 2, \mathtt{r} \mapsto 0, \mathtt{res} \mapsto 1, \mathtt{a} \mapsto 4, \mathtt{b} \mapsto 2]$

(a) Original program.

$[\mathtt{q} \mapsto \square, \mathtt{r} \mapsto \square, \mathtt{res} \mapsto \square, \mathtt{a} \mapsto 4, \mathtt{b} \mapsto 2]$

```
r := a;
while ( b <= r ) do {
  □;
  r := r - b
};
if ( ! (r = 0) )
then { □ }
else { res := 1 }
```

$[\mathtt{q} \mapsto \square, \mathtt{r} \mapsto \square, \mathtt{res} \mapsto 1, \mathtt{a} \mapsto \square, \mathtt{b} \mapsto \square]$

(b) Backward slice w.r.t. $\mathtt{res} \mapsto 1$.

Fig. 16: Slicing a program that computes whether b divides a.

To test whether 2 divides 4 we set $\mathtt{a} \mapsto 4$, $\mathtt{b} \mapsto 2$ in the input state. The remaining variables `q`, `r` and `res` are initialized to 0 (Figure 16a). The `while` loop body is executed twice; the loop condition is evaluated three times. Once the loop has stopped, variable `q` is set to 2 and variable `r` to 0. Since the `if` condition is false we execute the `else` branch and set `res` to 1. Figure 17 shows the execution trace.

(1) $\mathtt{r} := \mathtt{a}(4);$
(2) $\mathtt{while}_{\mathsf{true}}\ (\mathtt{b}(2) <= \mathtt{r}(4))\ \mathtt{do}\ \{$
(3) $\quad \mathtt{q} := \mathtt{q}(0) + 1;\ \mathtt{r} := \mathtt{r}(4) - \mathtt{b}(2)$
(4) $\};$
(5) $\mathtt{while}_{\mathsf{true}}\ (\mathtt{b}(2) <= \mathtt{r}(2))\ \mathtt{do}\ \{$
(6) $\quad \mathtt{q} := \mathtt{q}(1) + 1;\ \mathtt{r} := \mathtt{r}(2) - \mathtt{b}(2)$
(7) $\};$
(8) $\mathtt{while}_{\mathsf{false}}\ (\mathtt{b}(2) <= \mathtt{r}(0));$
(9) $\mathtt{if}_{\mathsf{false}}\ (\neg(\mathtt{r}(0) = 0))\ \mathtt{else}\ \{$
(10) $\quad \mathtt{res} := 1$
(11) $\}$

Fig. 17: Trace of executing an example program for $\mathtt{a} \mapsto 4$ and $\mathtt{b} \mapsto 2$.

We now want to obtain an explanation of `res`. We form a slicing criterion by setting $\mathtt{res} \mapsto 1$ (this is the value at the end of execution); all other variables are set to $\square$.

We begin by reconstructing the `if` conditional. We apply the second rule for $\mathtt{if}_{\mathsf{false}}$ slicing (Figure 16b). This is because c_2, i.e. the body of this branch, backward slices to an assignment `res := 1`, and not to a $\square$ (in which case the first rule for $\mathtt{if}_{\mathsf{false}}$ slicing would apply). Assignment in the `else` branch is

reconstructed by applying the second rule for assignment slicing. Since the value assigned to `res` is a constant it does not require presence of any variables in the state. Therefore state μ_a is empty. Moreover, variable `res` is erased in state μ; joining of μ_a and $\mu[\texttt{res} \mapsto \square]$ results in an empty state, which indicates that the code inside the `else` branch does not rely on the program state. However, to reconstruct the condition of the `if` we need a state μ_b that contains variable `r`. From the trace we read that $\texttt{r} \mapsto 0$, and so after reconstructing the conditional we have a state where $\texttt{r} \mapsto 0$ and all other variables, including `res`, map to $\square$.

We now apply the third rule for sequence slicing and proceed with reconstruction of the `while` loop. First we apply a trivial $\texttt{while}_{\textbf{false}}$ rule. The rule basically says that there is no need to reconstruct a `while` loop that does not execute – it might as well not be in a program. Since the final iteration of the `while` loop was reconstructed as a $\square$, we reconstruct the second iteration using the second $\texttt{while}_{\textbf{true}}$ backward slicing rule, i.e. the one where we have $T_w :: \mu \searrow \mu_w, \square$ as the first premise. We begin reconstruction of the body with the second assignment $\texttt{r} := \texttt{r}(2) - \texttt{b}(2)$. Recall that the current state assigns 0 to `r`. The RHS is reconstructed using the second rule for backward slicing of assignments we have already applied when reconstructing `else` branch of the conditional. An important difference here is that `r` appears both in the LHS and RHS. Reconstruction of RHS yields a state where $\texttt{r} \mapsto 2$ and $\texttt{b} \mapsto 2$ (both values read from a trace), whereas the current state contains $\texttt{r} \mapsto 0$. Here it is crucial that we erase `r` in the current state before joining. We apply third rule of sequence slicing and proceed to reconstruct the assignment to `q` using the first rule for assignment slicing (since $\texttt{q} \mapsto \square$ in the slicing criterion). This reconstructs the assignment as a $\square$. We then reconstruct the first iteration of the loop using the third $\texttt{while}_{\textbf{true}}$ slicing rule, since it is the case that $c_w \neq \square$. Assignments inside the first iteration are reconstructed following the same logic as in the second iteration, yielding a state where $\texttt{r} \mapsto 4$, $\texttt{b} \mapsto 2$, and other variables map to $\square$.

Finally, we reconstruct the initial assignment `r := a`. Since `r` is present in the slicing criterion, we yet again apply the second rule for assignment slicing, arriving at a partial input state $[\texttt{q} \mapsto 0, \texttt{r} \mapsto 0, \texttt{res} \mapsto 0, \texttt{a} \mapsto 4, \texttt{b} \mapsto 2]$ and a partial program shown in Figure 16b.

4 Formalization

In the previous sections we defined the syntax and semantics of the Imp language, and provided definitions of slicing in a Galois connection framework. We have implemented all these definitions in the Coq proof assistant [20] and proved their correctness as formal theorems. The following subsections outline the structure of our Coq development. We provide references to the source code by providing the name of file and theorem or definition as (`filename.v`: `theorem_name`, `definition_name`). We will use `*` in abbreviations like `*_monotone` to point to several functions ending with `_monotone` suffix. The whole formalization is around 5.2k lines of Coq code (not counting the comments). Full code is available online [19].

4.1 Lattices and Galois connections

Our formalization is built around a core set of definitions and theorems about lattices and Galois connections. Most importantly we define:

- that a relation that is reflexive, antisymmetric and transitive is a partial order (`Lattice.v`: `order`). When we implement concrete definitions of ordering relations we require a proof that these implementations indeed have these three properties, e.g. (`ImpPartial.v`: `order_aexpPO`).
- what it means for a function $f : P \to Q$ to be monotone (`Lattice.v`: `monotone`):
$$\forall_{x,y}\ x \sqsubseteq_P y \implies f(x) \sqsubseteq_Q f(y)$$
- consistency properties as given in Section 2.2 (`Lattice.v`: `inflation, deflation`).
- a Galois connection of two functions between lattices P and Q (see Definition 1 in Section 2.2) (`Lattice.v`: `galoisConnection`).

We then prove that:

- existence of a Galois connection between two functions implies their consistency and minimality (`Lattice.v`: `gc_implies_consistency, gc_implies_minimality`).
- two monotone functions with deflation and inflation properties form a Galois connection (`Lattice.v`: `cons_mono__gc`).

Throughout the formalization we operate on elements inside lattices of partial expressions ($\downarrow a, \downarrow b$, commands ($\downarrow c$) or states ($\downarrow \mu$). We represent values in a lattice with an inductive data type `prefix`[10] (`PrefixSets.v`: `prefix`) indexed by the top element of the lattice and the ordering relation[11]. Values of `prefix` data type store an element from a lattice together with the evidence that it is in the ordering relation with the top element. Similarly we define an inductive data type `prefixO` (`PrefixSets.v`: `prefixO`) for representing ordering of two elements from the same lattice. This data type stores the said two elements together with proofs that one is smaller than another and that both are smaller than the top element of a lattice.

4.2 Imp syntax and semantics

All definitions given in Figures 2–5 are directly implemented in our Coq formalization.

[10] Name comes from a term "prefix set" introduced in [17] to refer to a set of all partial values smaller than a given value. So a prefix set of a top element of a lattice denotes a set of all elements in that (complete) lattice.

[11] In order to make the code snippets in the paper easier to read we omit the ordering relation when indexing `prefix`.

Syntax trees for Imp (Figure 2), traces (Figure 6) and partial Imp (Figure 3) are defined as ordinary inductive data types in (`Imp.v`: `aexp, bexp, cmd`), (`Imp.v`: `aexpT, bexpT, cmdT`) and (`ImpPartial.v`: `aexpP, bexpP, cmdP`), respectively. We also define functions to convert Imp expressions to partial Imp expressions by rewriting from normal syntax tree to a partial one (`ImpPartial.v`: `aexpPartialize, bexpPartialize, cmdPartialize`).

Evaluation relations for Imp (Figures 7–9) and ordering relations for partial Imp (Figure 4) are defined as inductive data types with constructors indexed by elements in the relation (`Imp.v`: `aevalR, bevalR, cevalR` and `ImpPartial.v`: `aexpPO, bexpPO, comPO`), respectively. For each ordering relation we construct a proof of its reflexivity, transitivity, and antisymmetry, which together proves that a given relation is a partial order (`ImpPartial.v`: `order_aexpPO, order_bexpPO, order_comPO`).

Join operations (Figure 5) are implemented as functions (`ImpPartial.v`: `aexpLUB, bexpLUB, comLUB`). Their implementation is particularly tricky. Coq requires that all functions are total. We know that for two elements from the same lattice a join always exists, and so a join function is a total one. However, we must guarantee that a join function only takes as arguments elements from the same lattice. To this end a function takes three arguments: top element e of a lattice and two `prefix` values e_1, e_2 indexed by the top element e. So for example if e is a variable name x, we know that each of e_1 and e_2 is either also a variable name x or a $\square$. However, Coq does not have a built-in dependent pattern match and this leads to complications. In our example above, even if we know that e is a variable name x we still have to consider cases for e_1 and e_2 being a constant or an arithmetic operator. These cases are of course impossible, but it is the programmer's responsibility to dismiss them explicitly. This causes further complications when we prove lemmas about properties of join, e.g.:

$$e_1 \sqsubseteq e \wedge e_2 \sqsubseteq e \implies (e_1 \sqcup e_2) \sqsubseteq e$$

This proof is done by induction on the top element of a lattice, where e, e_1, and e_2 are all smaller than that element. The top element limits the possible values of e, e_1, and e_2 but we still have to consider the impossible cases and dismiss them explicitly.

4.3 Program state

Imp programs operate by side-effecting on a program state. Handling of the state was one of the most tedious parts of the formalization.

State is defined as a data type isomorphic to an association list that maps variables to natural number values (`ImpState.v`: `state`). Partial state is defined in the same way, except that it permits partial values, i.e. variables can be mapped to a hole or a numeric value (`ImpState.v`: `stateP`). We assume that no key appears more than once in a partial state. This is not enforced in the definition of `stateP` itself, but rather defined as a separate inductive predicate (`ImpState.v`: `statePWellFormed`) that is explicitly passed as an assumption to

any theorem that needs it. We also have a `statePartialize` function that turns a state into a partial state. This only changes representation from one data type to another, with no change in the state contents.

For partial states we define an ordering relation as a component-wise ordering of partial values inside a state (`ImpState.v`: `statePO`). This assumes that domains of states in ordering relation are identical (same elements in the same order), which allows us to formulate lemmas such as:

$$[\,] \leq \mu \implies \mu = [\,]$$

This lemma says that if a partial state μ is larger than an state with empty domain then μ itself must have an empty domain.

We also define a join operation on partial states, which operates element-wise on two partial states from the same lattice (`ImpState.v`: `stateLUB`).

As already mentioned in Section 3.1, the domain of the state is fixed throughout the execution. This means that updating a variable that does not exist in a state is a no-op, i.e. it returns the original state without any modifications. This behaviour is required to allow comparison of states before and after an update. Consider this lemma:

$$(\mu_1 \leq \mu_2) \wedge (v_1 \leq \mu_2(k)) \implies \mu_1[k \mapsto v_1] \leq \mu_2$$

It says that if a partial state μ_1 is smaller than μ_2 and the value stored in state μ_2 under key k is greater than v_1 then we can assign v_1 to key k in μ_1 and the ordering between states will be maintained. A corner-case for this theorem is when the key k is absent from the states μ_1 and μ_2. Looking up a non-existing key in a partial state returns a $\square$. If k did not exist in μ_2 (and thus μ_1 as well) then $\mu_2(k)$ would return $\square$ and so v_1 could only be a $\square$ (per second assumption of the theorem). However, if we defined semantics of update to insert a non-existing key into the state, rather than be a no-op, the conclusion of the theorem would not hold because domain of $\mu_1[k \mapsto v_1]$ would contain k and domain of μ_2 would not, thus making it impossible to define the ordering between the two states.

The approach described above is one of several possible design choices. One alternative approach would be to require evidence that the key being updated exists in the state, making it impossible to attempt update of non-existent keys. We have experimented with this approach but found explicit handling of evidence that a key is present in a state very tedious and seriously complicating many of the proofs. In the end we decided for the approach outlined above, as it allowed us to prove all the required lemmas, with only some of them relying on an explicit assumption that a key is present in a state. An example of such a lemma is:

$$\mu[k \mapsto v](k) = v$$

which says that if we update key k in a partial state μ with value v and then immediately lookup k in the updated state we will get back the v value we just wrote to μ. However, this statement only holds if k is present in μ. If it was absent the update would return μ without any changes and then lookup would

return $\square$, which makes the theorem statement false. Thus this theorem requires passing explicit evidence that $k \in \mathsf{dom}(\mu)$ in order for the conclusion to hold.

Formalizing program state was a tedious task, requiring us to prove over sixty lemmas about the properties of operations on state, totalling over 800 lines of code.

4.4 Slicing functions

Slicing functions implement rules given in Figures 10–15. We have three separate forward slicing functions, one for arithmetic expressions, one for logical expressions and one for imperative commands (`ImpSlicing.v`: `aexpFwd, bexpFwd, comFwd`). Similarly for backward slicing (`ImpSlicing.v`: `aexpBwd, bexpBwd, comBwd`).

In Section 2.2 we said that forward and backward slicing functions operate between two lattices. If we have an input p (an arithmetic or logical expression or a command) with initial state μ that evaluates to output p' (a number, a boolean, a state) and records a trace T then $\mathsf{fwd}^T_{(p,\mu)}$ is a forward slicing function parametrized by T that takes values from lattice generated by (p, μ) to values in lattice generated by p'. Similarly $\mathsf{bwd}^T_{p'}$ is a backward slicing function parametrized by T that takes values from lattice generated by p' to values in lattice generated by (p, μ). Therefore our implementation of forward and backward slicing functions has to enforce that:

1. (p, μ) evaluates to p' and records trace T
2. forward slicing function takes values from lattice generated by (p, μ) to lattice generated by p'
3. backward slicing function takes values from lattice generated by p' to lattice generated by (p, μ)

To enforce the first condition we require that each slicing function is parametrized by inductive evidence that a given input (p, μ) evaluates to p' and records trace T. We then define input and output types of such slicing functions as belonging to relevant lattices, which is achieved using the `prefix` data type described in Section 4.1. This enforces the conditions above. For example, the type signature of the forward slicing function for arithmetic expressions looks like this:

```
Fixpoint aexpFwd {st : state} {a : aexp}
   {v : nat} {t : aexpT}
   (ev : t :: a, st \\ v):
   (prefix a * prefix st) -> prefix v.
```

Here `ev` is evidence that arithmetic expression `a` with input state `st` evaluates to a natural number `v` and records an execution trace `t`. The `t :: a, st \\ v` syntax is a notation for the evaluation relation. The first four arguments to `aexpFwd` are in curly braces, denoting they are implicit and can be inferred from

the type of `ev`. The function then takes values from the lattice generated by `(a, st)` and returns values in the lattice generated by `v`.

In the body of a slicing function we first decompose the evaluation evidence with pattern matching. In each branch we implement logic corresponding to relevant slicing rules defined in Figures 10–15. Premises appearing in the rules are turned into recursive calls. If necessary, results of these calls are analysed to decide which rule should apply. For example, when backward slicing sequences we analyse whether the recursive calls return holes or expressions to decide which of the rules should apply.

The implementation of the slicing functions faces similar problems as the implementation of joins described in Section 4.2. When we pattern match on the evaluation evidence, in each branch we are restricted to concrete values of the expression being evaluated. For example, if the last step in the evaluation was an addition, then we know the slicing criterion is a partial expression from a lattice formed by expression $a_1 + a_2$. Yet we have to consider the impossible cases, e.g. having an expression that is a constant, and dismiss them explicitly. Moreover, operating inside a lattice requires us not to simply return a result, but also provide a proof that this result is inside the required lattice. We rely on Coq's `refine` tactic to construct the required proof terms. All of this makes the definitions of slicing functions very verbose. For example, forward slicing of arithmetic expressions requires over 80 lines of code with over 60 lines of additional boilerplate lemmas to dismiss the impossible cases.

For each slicing function we state and prove a theorem that it is monotone (`ImpSlicing.v`: `*_monotone`). For each pair of forward and backward slicing functions we state theorems that these pairs of functions have deflation and inflation properties (`ImpSlicing.v`: `*_deflating, *_inflating`), as defined in Section 2.2. Once these theorems are proven we create instances of a general theorem `cons_mono__gc`, described in Section 4.1, which proves that our definitions form a Galois connection and are thus a correctly defined pair of slicing functions. We also create instances of the `gc_implies_minimality` theorem, one instance for each slicing function. This provides us with a formalization of all the correctness properties, proving our main result:

Theorem 1. *Suppose* $\mu_1, c \Rightarrow T :: \mu_2$. *Then there exist total, monotone functions* $\mathsf{fwd}^T_{(c,\mu_1)} : \downarrow c \times \downarrow \mu_1 \to \downarrow \mu_2$ *and* $\mathsf{bwd}^T_{\mu_2} : \downarrow \mu_2 \to \downarrow c \times \downarrow \mu_1$. *Moreover,* $\mathsf{bwd}^T_{\mu_2} \dashv \mathsf{fwd}^T_{(c,\mu_1)}$ *form a Galois connection and in particular satisfy the minimality, inflation (consistency), and deflation properties.*

Here, the forward and backward slicing judgements are implemented as functions $\mathsf{fwd}^T_{(c,\mu_1)}$ and $\mathsf{bwd}^T_{\mu_2}$.

5 Related and Future Work

During the past few decades a plethora of slicing algorithms has been presented in the literature. See [18] for a good, although now slightly out of date, survey. Most of these algorithms have been analysed in a formal setting of some sort

using pen and paper. However, work on formalizing slicing in a machine checked way has been scarce. One example of such a development is [14], which formalizes dynamic slicing for π-calculus in Agda using a Galois connection framework identical to the one used in this paper. The high-level outline of the formalization is thus similar to ours. However, details differ substantially, since [14] formalizes a completely different slicing algorithm for concurrent processes using a different proof assistant. Another example of formalizing slicing in a proof assistant is [3], where Coq is used to perform an *a posteriori* validation of a slice obtained using an unverified program slicer. This differs from our approach of verifying correctness of a slicing algorithm itself. We see our approach of verifying correctness of the whole algorithm as a significant improvement over the validation approach. In a more recent work Léchenet et al. [9] introduce a variant of static slicing known as relaxed slicing and use Coq to formalize the slicing algorithm. Their work is identical in spirit to ours and focuses on the Imp language[12] with an extra `assert` statement.

Galois connections have been investigated previously as a tool in the mathematics of program construction, for example by Backhouse [1] and more recently by Mu and Oliveira [12]. As discussed in Section 1, Galois connections capture a common pattern in which one first specifies a space of possible solutions to a problem, the "easy" part, via one adjoint, and defines the mapping from problem instances to optimal solutions, the "hard" part, as the Galois dual. In the case of slicing, we have used the goal of obtaining a verifiable Galois connection, along with intuition, to motivate choices in the design of the forward semantics, and it has turned out to be easier for our correctness proof to define both directions directly.

Mechanized proofs of correctness of calculational reasoning has been considered in the Algebra of Programming in Agda (AOPA) system [11], and subsequently extended to include derivation of greedy algorithms using Galois connections [5]. Another interesting, complementary approach to program comprehension is Gibbons' *program fission* [7], in which the fusion law is applied "in reverse" to an optimized, existing program in order to attempt to discover a rationale for its behavior: for example by decomposing an optimized word-counting program into a "reforested" version that decomposes its behavior into "construct a list of all the words" and "take the length of the list". We conjecture that the traces that seem to arise as a natural intermediate structure in program slicing might be viewed as an extreme example of fission.

An important line of work on slicing theory focuses on formalizing different slicing algorithms within a unified theoretical framework of *program projection* [2]. Authors of that approach develop a precise definition of what it means that one form of slicing is weaker than another. However, our dynamic slicing algorithm does not fit the framework as presented in [2]. We believe that it should be possible to extend the program projection framework so that it can encompass slicing based on Galois connections but this is left as future work.

[12] Authors of [9] use the name WHILE, but the language is the same.

6 Summary

Program slicing is an important tool for aiding software development. It is useful when creating new programs as well as maintaining existing ones. In this paper we have developed and formalized an algorithm for dynamic slicing of imperative programs. Our work extends the line of research on slicing based on the Galois connection framework. In the presented approach slicing consists of two components: forward slicing, that allows to execute partial programs, and backward slicing, that allows to "rewind" program execution to explain the output.

Studying slicing in a formal setting ensures the reliability of this technique. We have formalized all of the theory presented in this paper using the Coq proof assistant. Most importantly, we have shown that our slicing algorithms form a Galois connection, and thus have the crucial properties of consistency and minimality. One interesting challenge in our mechanization of the proofs was the need to modify some of the theoretical developments so that they are easier to formalize in a proof assistant – c.f. overlapping rules for backward slicing of while loops described in Section 3.3.

Our focus in this paper was on a simple programming language Imp. This work should be seen as a stepping stone towards more complicated formalizations of languages with features like (higher-order) functions, arrays, and pointers. Though previous work [] has investigated slicing based on Galois connections for functional programs with imperative features, our experience formalizing slicing for the much simpler Imp language suggests that formalizing a full-scale langauge would be a considerable effort. We leave this as future work.

Acknowledgements

We gratefully acknowledge help received from Wilmer Ricciotti during our work on the Coq formalization, and Jeremy Gibbons for comments on a draft. This work was supported by ERC Consolidator Grant Skye (grant number 682315).

References

1. Backhouse, R.C.: Galois connections and fixed point calculus. In: Backhouse, R.C., Crole, R.L., Gibbons, J. (eds.) Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures. Lecture Notes in Computer Science, vol. 2297, pp. 89–148. Springer (2000). https://doi.org/10.1007/3-540-47797-7_4
2. Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, A., Korel, B.: A formalisation of the relationship between forms of program slicing. Science of Computer Programming **62**(3), 228–252 (2006), special issue on Source code analysis and manipulation (SCAM 2005)
3. Blazy, S., Maroneze, A., Pichardie, D.: Verified validation of program slicing. In: Proceedings of the 2015 Conference on Certified Programs and Proofs. pp. 109–117. CPP '15, ACM, New York, NY, USA (2015)

4. Cheney, J., Acar, U.A., Perera, R.: Toward a theory of self-explaining computation. In: In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman. pp. 193–216. No. 8000 in LNCS, Springer-Verlag (2013)
5. Chiang, Y., Mu, S.: Formal derivation of greedy algorithms from relational specifications: A tutorial. J. Log. Algebr. Meth. Program. **85**(5), 879–905 (2016). https://doi.org/10.1016/j.jlamp.2015.12.003
6. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press (2002)
7. Gibbons, J.: Fission for program comprehension. In: Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings. pp. 162–179 (2006). https://doi.org/10.1007/11783596_12
8. Korel, B., Laski, J.: Dynamic program slicing. Information Processing Letters **29**(3), 155–163 (1988)
9. Léchenet, J., Kosmatov, N., Gall, P.L.: Cut branches before looking for bugs: certifiably sound verification on relaxed slices. Formal Aspects of Computing **30**(1), 107–131 (Jan 2018)
10. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: Compcert – a formally verified optimizing compiler. In: ERTS 2016: Embedded Real Time Software and Systems. SEE (2016)
11. Mu, S., Ko, H., Jansson, P.: Algebra of programming in Agda: Dependent types for relational program derivation. J. Funct. Program. **19**(5), 545–579 (2009). https://doi.org/10.1017/S0956796809007345
12. Mu, S., Oliveira, J.N.: Programming from Galois connections. J. Log. Algebr. Program. **81**(6), 680–704 (2012). https://doi.org/10.1016/j.jlap.2012.05.003
13. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag, Berlin, Heidelberg (1999)
14. Perera, R., Garg, D., Cheney, J.: Causally consistent dynamic slicing. In: CONCUR. pp. 18:1–18:15 (2016)
15. Perera, R., Acar, U.A., Cheney, J., Levy, P.B.: Functional programs that explain their work. In: ICFP. pp. 365–376. ACM (2012)
16. Pierce, B.C., de Amorim, A.A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Software Foundations. Electronic textbook (2017), version 5.0. `http://www.cis.upenn.edu/~bcpierce/sf`
17. Ricciotti, W., Stolarek, J., Perera, R., Cheney, J.: Imperative functional programs that explain their work. Proceedings of the ACM on Programming Languages (PACMPL) **1**(ICFP) (Sep 2017)
18. Silva, J.: An analysis of the current program slicing and algorithmic debugging based techniques. Tech. rep., Technical University of Valencia (2008)
19. Stolarek, J.: Verified self-explaining computation (May 2019), `https://bitbucket.org/jstolarek/gc_imp_slicing/src/mpc_2019_submission/`
20. The Coq Development Team: The Coq proof assistant, version 8.7.0 (Oct 2017). https://doi.org/10.5281/zenodo.1028037
21. Weiser, M.: Program slicing. In: ICSE. pp. 439–449. IEEE Press, Piscataway, NJ, USA (1981)
22. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge, MA, USA (1993)