Systematic Design of Program Transformation Frameworks by Abstract Interpretation

Patrick Cousot and Radhia Cousot

Département d’informatique
École normale supérieure, 45 rue d’Ulm
75230 Paris cedex 05, France
cousot@ens.fr

Laboratoire d'informatique
École polytechnique
91128 Palaiseau cedex, France
rcousot@lix.polytechnique.fr

ABSTRACT

We introduce a general uniform language-independent framework for designing online and offline source-to-source program transformations by abstract interpretation of program semantics. Iterative source-to-source program transformations are designed constructively by composition of source-to-semantics, semantics-to-transformed semantics and semantics-to-source abstractions applied to fixpoint trace semantics. The correctness of the transformations is expressed through observational and performance abstractions. The framework is illustrated on three examples: constant propagation, program specialization by online and offline partial evaluation and static program monitoring.

1. INTRODUCTION

A program transformation is a meaning-preserving mapping defined on a programming language [21]. The program transformation methodology provides thinking tools for the development of programs from specifications (such as the fold/unfold transformation [24]) and program verification (such as temporal verification [9]). The program transformation techniques provide mechanical tools for program optimization (such as cache optimization [12], call-by-name to call-by-value transformation [26], constant propagation [18], continuation passing style transformation [23], deforestation [29], finite differencing [20], partial evaluation [1, 14], transition compression [16], software customization (such as security policy enforcement [10, 25], reverse engineering [31], slicing [30]) and compilation [22].

The objective of this paper is to introduce a general uniform language-independent framework for reasoning on semantics-based program transformation. The formalization is based on abstract interpretation [4, 6] which accounts for:

- the static program analyses that are used to justify transformations (such as binding time analysis [28] for partial evaluation [13, 15]);

- the correctness of transformations which should preserve the semantics, at some level of observation or abstraction of irrelevant details (this aspect is also standard and similar to the use of abstract interpretation to hierarchically organize programming language semantics [3]);

- the efficiency of transformations which should improve program performances as measured by an abstraction of the semantics (the use of abstract interpretation for performance analysis is more recent, see e.g. [19]);

- the formalization of syntactic program transformations as abstractions of program semantics transformations.

This last point is the main novelty of our approach which leads to a constructive language-independent program transformation design methodology where the syntactic transformation is constructed systematically by approximation of a semantic transformation which is easily shown to be correct and efficient. As noticed by [21], “Transformational systems may have the power to perform sophisticated program analysis and to generate software at breakneck speed, but to date they are not sound. Lacking from them is a convenient mechanical facility to prove that each transformation preserves semantics. In order to create confidence in the products of transformational systems we need to prove correctness of specifications and transformations. Currently, this is too labor-intensive to be practical.” The proposed uniform framework to formalize semantics-based program manipulation will hopefully be a useful step in that direction.

2. A FEW BASIC ELEMENTS OF ABSTRACT INTERPRETATION

Abstract interpretation [4, 6] formalizes the approximation correspondence between the concrete semantics $S[P]$ of a syntactically correct program $P \in P$, chosen in a given programming language $P$, and an abstract semantics $S\llbracket P \rrbracket$ which is a safe/conservative approximation of the concrete semantics $S[P]$.

The concrete semantics belongs to a concrete semantic domain $D$ which is a poset $(D, \sqsubseteq)$ when partially ordered by the approximation ordering $\sqsubseteq$ formalizing the loss of information (e.g. the logical implication). The abstract semantics is also a poset $(\mathcal{D}, \sqsubseteq)$ which is ordered by the abstract version $\sqsubseteq$ of the concrete approximation ordering $\sqsubseteq$. The concrete and abstract semantic domains often enjoy stronger properties, such as being complete partial orderings or complete lattices.
The correspondence between the concrete and the abstract semantic domains is given by a pair of maps \( \alpha \), which is the abstraction, and \( \gamma \), which is the concretization. The concretization \( \gamma(\mathcal{S}[P]) \) of the abstract semantics \( \mathcal{S}[P] \) expresses the abstract information available about program execution in concrete terms. It should be a sound approximation of the concrete semantics in that \( \mathcal{S}[P] \subseteq \gamma(\mathcal{S}[P]) \).

If any element \( S \) of the concrete domain \( po(D; \sqsubseteq) \) has a best approximation (i.e. \( \sqsubseteq \)-most precise) in the abstract domain \( po(\mathcal{D}; \sqsubseteq) \) given by \( \alpha(S) \), then the pair \( (\alpha, \gamma) \) is a Galois connection which is written \( po(D; \sqsubseteq) \xrightarrow{\alpha} po(\mathcal{D}; \sqsubseteq) \).

By definition, this means that \( \forall X \in D; \forall Y \in \mathcal{D}; \alpha(X) \sqsubseteq Y \Leftrightarrow X \sqsubseteq \gamma(Y) \). A Galois insertion \( po(D; \sqsubseteq) \xrightarrow{\alpha} po(\mathcal{D}; \sqsubseteq) \) is a Galois connection with \( \alpha \) surjective.

One interest of the abstract interpretation theory is to constructively derive the exact (resp. approximate) abstract semantics \( \mathcal{S}[P] \) from the given concrete semantics \( \mathcal{S}[P] \) by refining the specification \( \alpha(\mathcal{S}[P]) = \mathcal{S}[P] \) (resp. \( \alpha(\mathcal{S}[P]) \subseteq \mathcal{S}[P] \)). If e.g. the concrete semantics is given in fixpoint form \( \mathcal{S}[P] = \text{lfp}_\mathcal{D}(F[\mathcal{P}]) \) where the semantic transformer \( F[\mathcal{P}] \) is monotonic, then the abstract semantics can be chosen as \( \text{lfp}_\mathcal{D}(F[\mathcal{P}]) \) where the abstract semantic transformer \( F[\mathcal{P}] \) is designed using the local commutation conditions \( \alpha \circ F[\mathcal{P}] \circ \gamma = F[\mathcal{P}] \circ \alpha \gamma \). Several fixpoint transfer theorems e.g. [6, Th. 7.1.0.4(3)], [3, Th. 2.1], [8, Cor. 2.4] (resp. fixpoint upper approximation theorems e.g. [6, Th. 7.1.0.4(2)], [8, Th. 2.5]) guarantee that:

\[
\alpha(\text{lfp}_\mathcal{D}(F[\mathcal{P}])) = \text{lfp}_\mathcal{D}(F[\mathcal{P}] \circ \alpha \gamma) \quad (\text{resp. } \alpha(\text{lfp}_\mathcal{D}(F[\mathcal{P}])) \subseteq \text{lfp}_\mathcal{D}(F[\mathcal{P}])) \quad (1)
\]

and their duals (with \( \sqsubseteq \), \( \sqsupseteq \), \( \gamma \) substituted for \( \sqsubseteq \), \( \sqsupseteq \), and \( \alpha \)).

3. PRINCIPLE OF PROGRAM TRANSFORMATION

In this section we introduce the principle of our formalization of program transformations by abstract interpretation.

3.1 Syntactic Program Transformation

A syntactic program transformer \( t \) takes as input a subject program \( \mathcal{P} \in \mathcal{P} \) and, upon termination, produces as output a transformed program \( t[\mathcal{P}] \in \mathcal{P}' \) (for short \( \mathcal{P}' = \mathcal{P} \)).

\[
\begin{array}{c}
\text{Subject program } \mathcal{P} \in \mathcal{P} \\
\text{Syntactic transformation } t \\
\text{Transformed program } t[\mathcal{P}] \in \mathcal{P}'
\end{array}
\]

However, program transformations seldom rely on syntactic criteria only, so we now introduce the semantic foundations which are especially needed to determine meaning preservedness of program transformations. We first consider online program transformations directly referring to program executions and next offline transformations based upon a preliminary static program analysis.

3.2 Semantics-Based Online Program Transformation

Online transformations refer to program executions. For example, online partial evaluation makes use of the concrete values of the static input variables so that, according to [16, Def. 4.6], the concrete values computed during program specialization can affect the choice of action taken.

Formally, an online transformation can be understood as making use of the program semantics \( \mathcal{S}[\mathcal{P}] \in \mathcal{D} \). From this formal point of view, any program transformer \( t \in \mathcal{P} \rightarrow \mathcal{P} \) on the program syntax induces a corresponding semantic transformer \( t \in \mathcal{D} \rightarrow \mathcal{D} \) taking as input the semantics \( \mathcal{S}[\mathcal{P}] \) of the subject program \( \mathcal{P} \) and producing the semantics \( t[\mathcal{S}[\mathcal{P}]] \) of the transformed program \( t[\mathcal{P}] \). A strong equivalence requirement is that \( \mathcal{S}[t[\mathcal{P}]] = t[\mathcal{S}[\mathcal{P}]] \) stating that the semantics of the syntactically transformed program is precisely the semantic transformation of the semantics of the subject program:

\[
\begin{array}{c}
\text{Subject program } \mathcal{P} \\
\text{Syntactic transformation } t \\
\text{Transformed program } t[\mathcal{P}]
\end{array}
\]

\[
\begin{array}{c}
\text{Semantics } \mathcal{S} \\
\text{Semantics } \mathcal{S}
\end{array}
\]

A generalization consists in considering \( \mathcal{P} \) as a tuple of programs, see Sec. 8. We now study in more details the correspondences between the various elements of this diagram.

3.3 Correspondence Between Syntax and Semantics of Programs

Programs can be considered as an abstraction of their semantics. For example the syntax of programs records the existence of variables and maybe their type but not the sequence of their successive values during execution, as defined by the semantics. Usually programs record the chaining of actions but not their exact sequences of execution. The same way, program performances are completely abstracted in the program syntax although execution time might be included in the operational semantics. Formally:

*po(D; \sqsubseteq) <\!\!\!\!\rightarrow_{\alpha} po(\mathcal{D}; \sqsubseteq)/p

where \( \mathcal{S}[\mathcal{P}] \) is the semantics of program \( \mathcal{P} \in \mathcal{P} \) while \( p[S] \) is the simplest program whose semantics upper-approximates \( S \in \mathcal{D} \). Programs are considered up to a syntactic equivalence \( \mathcal{P} \equiv \mathcal{P} \) means “is defined as”.

The syntactic refinement is \( p \sqsubseteq \mathcal{P} \subseteq (S[p] \sqsubseteq S[q]) \). In practice, neither \( \sqsubseteq \) nor \( \equiv \) are computable and they must be approximated (e.g. by \( = \) and \( \subseteq \) in Sec. 6.6).

3.4 Syntactic Program Transformations as Abstractions of Semantics-Based Program Transformations

Thanks to the above correspondence between the syntax and semantics of programs, the transformed program \( t[\mathcal{P}] \) can be viewed as the decomposition \( p[t[S[\mathcal{P}]]] \) of its semantics \( t[S[\mathcal{P}]] \). Using this correspondence \( (S, p) \) between the syntax and the semantics of a program as well as the semantic form \( t \) of the program syntactic transformation \( t \), we get the following commuting schema (dashed arrows are unused in the explanation):

\[
\begin{array}{c}
\text{Subject program } \mathcal{P} \\
\text{Syntactic transformation } t \\
\text{Transformed program } t[\mathcal{P}]
\end{array}
\]

\[
\begin{array}{c}
\text{Subject program semantics } S[\mathcal{P}] \\
\text{Semantic transformation } t \\
\text{Transformed program semantics } t[S[\mathcal{P}]]
\end{array}
\]
this schema leads to another view of online program transformation. A syntactic program transformation algorithm \( t[P] = p[t[S[P]]] \) is derived by abstraction of its semantic specification \( t[S[P]] \). In practice the syntactic transformation \( t[P] \) may be weaker, that is more restricted or less effective, than the ideal semantics-based but undecidable transformation \( p[t[S[P]]] \) so that \( t[P] \oplus p[t[S[P]]] \).

3.5 Correspondence Between Syntactic and Semantic Program Transformations

Formally, the semantic transformation \( t \in \mathcal{D} \rightarrow \mathcal{D} \) induced by a syntactic transformation \( t \in \mathcal{P} \rightarrow \mathcal{P} \) is:

\[
\text{t} \in \mathcal{D} \rightarrow \mathcal{D} \quad t[S] = S[t[S[P]]].
\]

Conversely, the syntactic transformation \( t \in \mathcal{P} \rightarrow \mathcal{P} \) induced by a semantic transformation \( t \in \mathcal{D} \rightarrow \mathcal{D} \) is:

\[
t[P] = p[t[S[P]]].
\]

We use (3) as a basis for designing the syntactic transformation \( t[P] \) formally from the semantic transformation \( t[S[P]] \). Observe that \( p: \mathcal{D} \rightarrow \mathcal{P} \) implies

\[
\text{po}(\mathcal{D}; \sqsubseteq) \xrightarrow{p} \text{po}(\mathcal{P}; \sqsubseteq).
\]

so that for \( \sqsubseteq \)-monotonic transformations, the source-to-source syntactic transformation is a functional abstraction of the semantic transformation. However, because of undecidability, we can often only compute an effective approximation \( t[P] \approx p[t[S[P]]] \).

3.6 Correspondence Between the Subject and Transformed Program Semantics

A program transformation corresponds to a loss of information on the subject program whence its semantics. For example, in partial evaluation, the transformed program is specialized for given static values so that the information on how these static values are computed in the subject program is lost in the specialization process. In the abstract interpretation framework, this can be formalized as:

\[
\text{po}(\mathcal{D}; \sqsubseteq) \xrightarrow{t} \text{po}(\mathcal{D}; \sqsubseteq).
\]

By composition of the Galois connections (2) and (4), by definition (3) and by \( p \circ S[P] = \text{po}(\mathcal{P}; \sqsubseteq) \) we have:

\[
\text{po}(\mathcal{P}; \sqsubseteq) \xrightarrow{t} \text{po}(\mathcal{P}; \sqsubseteq).
\]

When the fixpoint \( \text{lfp } F[P] \) is defined on posets satisfying the ascending chain condition, this fixpoint characterization \( t[P] = \text{lfp } F[P] \) of the syntactic program transformation \( t[P] \) directly leads to an iterative algorithm.

However, the semantic transformation \( t \) can depend on undecidable results on the program semantics so that the syntactic transformation \( t[P] \) may not terminate or, even worse, fixpoint transfer theorems (1) may not be applicable. In this case, weaker approximate transformations must be considered which depend upon decidable criteria only. This leads to the idea of offline program transformation considered in Sec. 3.10. But before, we study semantics-based correctness criteria of program transformation.

3.8 Correctness of Online Program Transformation: Observational Abstraction

Another advantage of understanding syntactic program transformation as an abstraction of a semantics transformation is that it naturally leads to a simple notion of correctness of the program transformation. A transformation is correct if, at some level of abstraction, the observation of the execution of the subject program is equivalent to the observation of the execution of the transformed program:

\[
\text{Subject program } P \quad \quad \text{Syntactic transformation } t \quad \quad \text{Transformed program } t[P]
\]

\[
\text{Semantics } S \quad \quad \text{Observational semantics } \alpha \quad \quad \text{Transformed program semantics } S[t[P]]
\]

Observational equivalence can be formalized in the abstract interpretation framework by requiring the abstraction of the semantics of the subject and of the transformed programs to be identical. The specification of the observational abstraction \( \alpha \) should be considered part of the problematics. For example, BT ignoring the termination problem [16, Ch. 4, note on p. 69] can be formalized by an observational abstraction ignoring all infinite program behaviors. Schematically:

\[
\text{Subject program semantics } S[P] \quad \quad \text{Observational equivalence} \quad \quad \text{Transformed program semantics } S[t[P]]
\]

Formally, a source-to-source program transformation \( t \in \mathcal{P} \rightarrow \mathcal{P} \) is said to be \textit{correct with respect to an observational abstraction}:

\[
\text{po}(\mathcal{D}; \sqsubseteq) \xrightarrow{\alpha} \text{po}(\mathcal{D}; \sqsubseteq) \quad \quad \quad (6)
\]

if and only if for all programs \( P \in \mathcal{P} \), \( \alpha \circ S[P] = \alpha \circ S[t[P]] \), as shown in diagramed form, in Fig. 1. More generally, programs \( P \) and \( Q \) are said to be \( \alpha \)-observationally equivalent, written \( P \equiv_\alpha Q \), if and only if:

\[
\alpha \circ S[P] = \alpha \circ S[Q].
\]

3.9 Principle of Online Program Transformation

Summarizing this point of view on online program transformation, we get the schematic diagram of Fig. 2 including a formalization of the transformation correctness through a
syntactic observational abstraction $\alpha_\circ$. According to this diagram, the syntactic transformation $t[P]$ is designed as an upper-approximation of the best possible one $p[t[S[P]]]$. This consists in simplifying the term $p[t[S[P]]]$ in order to get rid of all semantic subterms in $S$ and $t$.

By definition of the Galois connection (2), the correctness condition $p[t[S[P]]] \sqsubseteq t[S[P]]$ is equivalent to $t[S[P]] \sqsubseteq t[S[t[P]]]$. This equivalence leads to an alternative method for designing the syntactic transformation $t$. Starting from the given semantic transformation $t$, the term $t[S[P]]$ is simplified by pushing out $S$ in order to rewrite it in the form $S[t[S[P]]]$ so as to extract the definition of $t[S[P]]$. This methodology, which is quite common in abstract interpretation (e.g. [3]), is illustrated in Sec. 7.1.4.

In both cases, the transformation is an upper-approximation and so must be proved correct. To do so we prove that $S[P] \sqsubseteq t[S[P]]$ and $\alpha_\circ(S[P]) \sqsubseteq \alpha_\circ(t[S[P]])$. Then, by (6), $\alpha_\circ$ is monotonic so $\alpha_\circ(S[P]) \sqsubseteq \alpha_\circ(t[S[P]]) \sqsubseteq \alpha_\circ(t[S[t[P]]])$. By antisymmetry, we conclude that $\alpha_\circ(S[P]) = \alpha_\circ(t[S[P]]) = \alpha_\circ(t[S[t[P]]]).$

3.10 Principle of Offline Program Transformation

Although offline program transformation does not directly use the values of variables during program execution, it nevertheless uses some information on program execution, which is obtained by a preliminary static program analysis. For example, in partial evaluation, a preliminary binding time analysis is used to compute a division of program variables into static and dynamic ones, the transformation being only applied to static ones [16]. From a syntactic point of view, the preliminary static analysis phase is used to add annotations to the program and then the transformation is applied to the annotated program. This leads to the schema of Fig. 3, which is the schema given for online program transformation in Sec. 3.9, but for the fact that it is applied to an annotated program $P$ derived from the subject program $P$ by a preliminary static analysis based annotation algorithm. Although the preliminary syntactic annotation phase can be very useful as a user interface, one can isolate the preliminary program static phase as an abstract interpretation of the program semantics as specified by a static analysis Galois connection

$$\text{po}(P; \sqsubseteq) \xrightarrow{\sqsubseteq \circ} \text{po}(\overline{S}; \sqsubseteq)$$

and consider syntactic transformations acting on the program $P$ given its abstract semantics $\overline{S}[P]$, as shown in Fig. 4. We now exemplify this framework on elementary program transformations of a simple imperative programming language.

3.11 Transformation Combinations

Since the composition of Galois connections is a Galois connection, the transformation diagrams of Fig. 2, Fig. 3 and Fig. 4 can be combined serially or in parallel (for multi-programs transformations as exemplified in Sec. 8) to explain complex combinations of transformations. For example, the reduced product [6] of constant propagation (Sec. 6) and online partial evaluation (Sec. 7.1) leads to an offline partial evaluator where the values of some of the static variables is detected by the preliminary constant detection analysis (Sec. 6.2).

4. SYNTAX AND SEMANTICS OF THE EXAMPLE PROGRAMMING LANGUAGE

Let us consider imperative iterative programs acting on global variables such as e.g.

$$X := \text{?}; \ \text{while } X > 0 \ \text{do } X := X + 1 \ \text{od}$$

that would be written:

$$a: \ X := \text{?} \to b; \ \ \ c: \ X := X + 1 \to d;$$
$$b: \ (X > 0) \to c; \ \ \ d: \ \text{skip} \to b;$$
$$b: \ (X > 0) \to e; \ \ \ e: \ \text{stop};$$

in the example language $P$. If execution is at some label $L$ then one of the transitions $L: A \to L'$; labeled with $L$ is executed, provided the action $A$ is not blocking and the execution can go on by branching to the next label $L'$. Programs are nondeterministic since several actions can be referenced by the same label. If no action is labelled $L'$, the execution is blocked at $L$, which is the case for the $\text{stop}$ command $L: \text{stop}$, which is a shorthand for $L: \text{skip} \to L'$; where $L'$ is the undefined label. The $\text{skip}$ command $L: \text{skip} \to L$ is itself a shorthand for the boolean test $L: \text{true} \to L$.

Formally, programs are not restricted to be finite. This is useful to discuss program transformations such as partial evaluation which may not terminate. However, in practice, program transformations are required to be effective so as to produce finite transformed programs out of finite subject programs.
4.1 Abstract Syntax of Programs

The abstract syntax of programs is defined in Fig. 5. We let \( \text{var}[D] \) be the set of variables of an expression or action \( D \in E \cup B \cup A \) and define:

\[
\begin{align*}
\text{lab}[L_1 : A \rightarrow L_2 ;] & \triangleq L_1, \quad \text{var}[L_1 : A \rightarrow L_2 ;] \triangleq \text{var}[A], \\
\text{act}[L_1 : A \rightarrow L_2 ;] & \triangleq A, \quad \text{lab}[P] \triangleq \{ \text{lab} \{C| C \in P\} \}, \\
\text{suc}[L_1 : A \rightarrow L_2 ;] & \triangleq L_2, \quad \text{var}[P] \triangleq \bigcup_{C \in P} \text{var}[C].
\end{align*}
\]

A stop command is \( L : \text{stop} \); \( L : \text{ski} p \rightarrow 1 \); and a ski \( p \) command is \( L : \text{ski } p \rightarrow L; \) \( L : \text{true} \rightarrow L'. \)

4.2 Environments

The commands of a program \( P \) act on global variables \( X \in \text{var}[P] \). The program variables take their value in the semantic domain \( \mathbb{D} \). This value can be the uninitalized or undefined value \( \mathbb{U} \not\in \mathbb{D} \) so \( \mathbb{U}_0 \triangleq \mathbb{D} \cup \{\mathbb{U}\} \). An environment \( \rho \in \mathcal{E} \) maps variables \( X \in \text{dom}[\rho] \) to their value \( \rho(X) \) so \( \mathcal{E} \triangleq \bigcup_{X \subseteq X} \mathcal{E}[X] \) where \( \mathcal{E}[X] \triangleq X \rightarrow \mathbb{D}_0 \) is the subset of environments \( \rho \) with \( \text{dom}[\rho] \subseteq X \). In particular the empty environment \( \rho \) is the only environment with empty domain \( \text{dom}[\rho] = \emptyset \) so that \( \mathcal{E}[\emptyset] \triangleq \emptyset \rightarrow \mathbb{D}_0 = \{\emptyset\} \). \( \mathcal{E}[P] \) is

\[
\text{the set of environments of a program } P \text{ whose domain is the set of program variables: } \mathcal{E}[P] \triangleq \mathcal{E}[\text{var}[P]].
\]

\( \rho|_{X} \) where \( X \subseteq X \) is the restriction of environment \( \rho \) to the domain \( \text{dom}[\rho] \cap X \). We write \( \rho \setminus X \) for the restriction of environment \( \rho \) to the variables not in \( X \). Therefore \( \rho \setminus X \) is the environment \( \rho' \) such that \( \text{dom}[\rho'] = \{X \in \text{dom}[\rho]|X \not\subseteq X\} \) and \( \forall X \in \text{dom}[\rho']: \rho'(X) = \rho(X) \).

4.3 Semantics of Program Actions

The semantics \( A[E] \) of an arithmetic expression \( E \) is defined inductively\(^2\):

\[
A[n] \rho \triangleq n, \quad A[X] \rho \triangleq \rho(X), \quad A[E_1 - E_2] \rho \triangleq A[E_1] \rho - A[E_2] \rho
\]

where \( - \) is extended to the undefined value \( \mathbb{U} \) as \( \mathbb{U} - \mathbb{U} \triangleq \mathbb{U} - n \triangleq n - \mathbb{U} \triangleq \mathbb{U} \). \( A[E] \in \mathcal{E}[X] \) is well-defined when \( \text{var}[E] \subseteq X \) whence \( A[E] \in \mathcal{E}[P] \) is well-defined when the arithmetic expression \( E \) belongs to a program \( P \). Similarly, we define the semantics \( B[B] \in \mathcal{E}[X] \).

\(^2\) For simplicity, here and afterwards, we do not distinguish between values \( n = \text{val}(n) \) and their denotations \( n = \text{str}(n) \) (including for booleans so \( \text{true} \) denotes \( \text{true} \)). In particular the denotation of the undefined value \( \mathbb{U} \) is also written \( \mathbb{U} \) but should be some uninitalized variable \( \text{undefined} \).
\[ E ::= n \quad \text{Integer numbers} \\
X ::= X \quad \text{Program variables} \\
E ::= E \quad \text{Arithmetic expressions} \\
E ::= n \quad \text{Integer} \\
| X \quad \text{Variable} \\
| E_1 - E_2 \quad \text{Difference} \\
B ::= B \quad \text{Boolean expressions} \\
B ::= E_1 < E_2 \quad \text{Comparison} \\
| B_1 \lor B_2 \quad \text{Disjunction} \\
| \neg B_1 \quad \text{Negation} \\
| \text{true} \quad \text{true} \\
A ::= A \quad \text{Program actions} \\
A ::= X := E \quad \text{Assignment} \\
| X := ? \quad \text{Random assignment} \\
| B \quad \text{Test} \\
L ::= L \quad \text{Program labels} \\
\text{l} \notin L \quad \text{Undefined label} \\
L ::= L \circ L \quad \text{Colabels} \\
C ::= C \quad \text{Commands} \\
C ::= L_1 : A \rightarrow L_2 \quad \text{Transition command} \\
P ::= P \quad \text{Programs} \\
\text{Figure 5: Abstract syntax of programs} \\
\]

4.5 Partial Trace Semantics of Programs

We let \( \Omega \) be the set of finite partial traces. \( \Omega \) is therefore defined as the set of all sequences \( \sigma \) of states of length \( \#\sigma \geq 0 \) such that any state \( \sigma_i, i \in [1, \#\sigma] \) the trace is a possible successor of the previous state \( \sigma_{i-1} \). \( \sigma_i \in S(\sigma_{i-1}) \). The set \( S^*[P] \subseteq \Omega \) of finite partial traces of a program \( P \) in is, in fixpoint form, \( S^*[P] = \text{lfp}^* F^*[P] \) where, for the backward case:

\[ F^*[P]T \triangleq \Theta[P] \cup \{ S' \sigma \mid S' \in S[P](S \wedge S \sigma \in T) \}, \]

and similarly for the forward case. If we only are interested in those executions of a program \( P \) starting from a given set \( \Theta[P] \) of entry points so that \( \Theta[P] \triangleq \{ (\rho, C) \mid \rho \in E \wedge \text{var}[P] \subseteq \text{dom}[\rho] \wedge C \in P \land \text{lab}[C] \in E[P] \} \) is the set of initial states, we can consider the partial trace semantics \( S^*[P] \subseteq \Omega \) of \( P \) which is the set of partial traces \( \sigma \in \Omega \) starting from an initial state \( \sigma_0 \in \Theta[P] \). The partial trace semantics \( S^*[P] \) can be expressed in fixpoint form as \( \text{lfp}^* F^*[P] \) where:

\[ F^*[P]T \triangleq \Theta[P] \cup \{ S^* \sigma \mid S^* \in S[P](S \wedge S \sigma \in T) \} \]

4.6 Correspondence Between Program Syntax and Semantics

The trace semantics maps programs to sets of traces. Conversely, we map sets of traces to programs by collecting commands executed along traces so \( p^* \in p(\Omega) \mapsto P \) is:

\[ p^*[T] \triangleq \{ C \mid \exists \sigma \in T : \sigma_i \in [0, \#\sigma] \land p \in E : \sigma_i = (\rho, C) \} . \]

Following (2), we have a Galois connection:

\[ \text{po}(p^*); \quad \text{po}(P) \]

where \( PF \) is the quotient of \( P \) by the syntactic equivalence \( P \equiv Q \) and \( P \equiv Q \) is the syntactic refinement. The Galois connection follows from \( p^* \) and \( S^*[P] \) are monotonic, \( p^* \circ S^*[P] \) is \( P \) up to dead code elimination and equivalences such as \( L : \text{stop} \in L : \text{skip} \mapsto L \); so \( p^* \circ S^*[P] \) was \( P \) and \( T \subseteq S^*[P] \) since all commands along the traces of \( T \) are collected.

5. ACTION SPECIALIZATION

5.1 Definition of Expression Specialization

The residual of an arithmetic or boolean expression \( E \) in a given (so-called “static”) environment \( \rho \) (assigning “static” values to the “static” variables \( \text{var}[\rho] \)) is the expression \( R[E]\rho \) resulting from the specialization of that expression \( E \) to that environment \( \rho \). Expression specialization is defined in Fig. 6 (where values and their denotations are once again confounded).

An expression \( D \in E \cup B \) is static in the (so-called “static”) environment \( \rho \) (written \( \text{static}[D]\rho \)) if and only if it can be fully evaluated in this environment \( \rho \), that is \( \text{var}[D] \subseteq \text{dom}[\rho] \). Otherwise \( \text{var}[D] \not\subseteq \text{dom}[\rho] \) and the expression \( D \) is dynamic in the environment \( \rho \).

The specialization of an arithmetic expression \( E \in E \) which is static in an environment \( \rho \) always yields a static value (i.e. a constant): \( \text{static}[E]\rho = (R[E]\rho) \in B_0 \) and similarly for boolean expressions \( E \in B : \text{static}[B]\rho = (R[B]\rho) \in B_0 \).

---

\( ^* \) For short we exclude infinite traces. This is not a problem for the considered safety-preserving example program transformations because if sets of prefix-closed finite traces are observationally equivalent then so is their limit. Otherwise, see [8].
\[ R \in \mathcal{E} \xrightarrow{\rho} \mathcal{E} \xrightarrow{\rho} \mathcal{E} \]
\[ R[x] \rho \triangleq \text{if } x \in \text{dom} (\rho) \text{ then } \rho(x) \text{ else } \}
\[ R[E_1 \cdot E_2] \rho \triangleq \text{let } E'_1 = R[E_1] \rho \text{ and } E'_2 = R[E_2] \rho \text{ in}
\text{if } E'_1 = \emptyset \text{ or } E'_2 = \emptyset \text{ then } \emptyset
\text{elsif } E'_1 = n_1 \text{ and } E'_2 = n_2 \text{ and } n = n_1 - n_2
\text{then } n \text{ else } E'_1 - E'_2 \]
\[ R \in \mathcal{E} \xrightarrow{\rho} \mathcal{E} \xrightarrow{\rho} \mathcal{E} \]
\[ R[E_1 < E_2] \rho \triangleq \text{let } E'_1 = R[E_1] \rho \text{ and } E'_2 = R[E_2] \rho \text{ in}
\text{if } E'_1 = \emptyset \text{ or } E'_2 = \emptyset \text{ then } \emptyset
\text{elsif } E'_1 = n_1 \text{ and } E'_2 = n_2 \text{ and } b = n_2 < n_2
\text{then } b \text{ else } E'_1 < E'_2 \]
\[ R[b_1 \or b_2] \rho \triangleq \text{let } B'_1 = R[b_1] \rho \text{ and } B'_2 = R[b_2] \rho \text{ in}
\text{if } B'_1 = \emptyset \text{ then } \emptyset
\text{elsif } B'_1 = \text{true} \text{ or } B'_2 = \text{true} \text{ then } \text{true}
\text{elsif } B'_1 = \text{false} \text{ then } B'_2
\text{elsif } B'_2 = \text{false} \text{ then } B'_1
\text{else } B'_1 \or B'_2 \]
\[ R[-b] \rho \triangleq \text{let } b' = R[b] \rho \text{ in}
\text{if } b' = \emptyset \text{ then } \emptyset
\text{elsif } b' = \text{true} \text{ then } \text{false}
\text{elsif } b' = \text{false} \text{ then } \text{true}
\text{else } -b' \]
\[ R[\text{true}] \rho \triangleq \text{true} \quad R[\text{false}] \rho \triangleq \text{false} \]

Figure 6: Expression specialization

5.2 Correctness of Expression Specialization

Intuitively, expression specialization is correct in that the semantics of the residual expression restricted to dynamic variables is equivalent to the semantics of the subject expression with the same static variables. Formally, for any arithmetic expression \( E \), any (so-called “static”) environment \( \rho \) and any (so-called “dynamic”) environment \( \rho' \) such that \( \rho \not\subseteq \rho' \), we have \( A[R[E] \rho'] = A[E] \rho' \). Moreover the evaluation of the residual only depends on the dynamic variables in that \( A[R[E] \rho'] = A[E] \rho' \). The residual expression \( E \) in the environment \( \rho \) (such that \( \text{static}(E) = \rho \)), the residual is equal (up to the value/denotation correspondence) to the classical evaluation of Sec. 4.3: \( R[E] \rho = A[E] \rho \). Similar results hold for boolean expressions \( B \).

5.3 Definition of Action Specialization

The specialization \( R[A] \rho \) of an action \( A \) in an environment \( \rho \) is defined in Fig. 7. Action specialization produces both a residual environment and a residual action, the residual environment possibly recording the value of static variables (upon initialization or e.g. after assignment of a constant) or no longer recording the value of dynamic variables (e.g. after a random assignment).

Action specialization is an abstraction in that (representing properties as usual by the set of elements having this property):
\[ \text{po}(\rho(E \times A); \subseteq) \xrightarrow{\gamma^R} \text{po}(\rho(E \times A); \subseteq) \]
where \( \alpha_X(X) \triangleq \{ R[A] \rho \mid (\rho, A) \in X \} \)

5.4 Correctness of Action Specialization

The specialization \( (\rho, A_\rho) = R[A] \rho_\theta \) of an action \( A \) in a (so-called “static”) environment \( \rho_\theta \) is correct since the residual action \( A_\rho \) and subject action \( A \) have the same semantics in environments with identical static variables, \( \rho_\theta \). The new static environment after execution of action \( A \) and the execution of the residual action \( A_\rho \) depends on dynamic variables only. Formally, for all actions \( A \), if \( (\rho, A_\rho) = R[A] \rho \) and \( \rho_0 \in \rho' \) then \( S[\rho_0] A \rho' = S[A] \rho' \). Moreover, \( \rho' \in S[A] \rho' \). The new environment \( \rho' \) is the new static environment after execution of action \( A \).

6. EXAMPLE 1: CONSTANT PROPAGATION

1. Observational Abstraction

The observational abstraction for constant propagation gets rid of commands but preserves the sequence of environments observed along a partial trace:

\[ (\alpha_\sigma^*(T)) \triangleq \{ (\alpha_\sigma^*(\sigma)) \mid \sigma \in T \}, \]
\[ (\alpha^*(\sigma)) \triangleq \lambda \alpha : (\alpha^*(\sigma)) \triangleq \rho . \]

It is therefore insensitive to the modification of the program actions, relabelling (contrary to \( \eta \)) and dead code elimination (like \( \eta \)). We have:
\[ \text{po}(\rho(\not\in T); \subseteq) \xrightarrow{\gamma^T} \text{po}(\rho(\not\in T); \subseteq) \]
where \( \not\in T \) is the set of finite sequences of environments.

1. Constant Detection Analysis

Constant detection static analysis \( S'[P] \) of a program \( P \) [18] is a sound abstract interpretation \( \alpha' : S' [P] \subseteq S'[P] \) of the program partial trace semantics \( S'[P] \alpha \) [6] for the following abstraction (which is the upper adjoint of the Galois connection (8)):
\[ \alpha'(T) \triangleq \lambda \mathcal{L} \cdot \lambda \mathcal{X} \cdot \bigcup \{ (\rho(X); \exists \sigma : \exists C : \exists i ; \sigma_i = (\rho, C) \wedge \text{lab}[C] = \mathcal{L} \} \]
where \( \bigcup \) is the pointwise extension of the least upper bound \( \bigcup \) in the complete lattice \( \mathcal{D}^* \triangleq \mathcal{D}_0 \cup \{ \bot, T \} \) partially ordered by \( \forall X \in \mathcal{D}^* : \bot \subseteq X \subseteq X \subseteq T \).
6.3 Offline Semantic Constant Propagation

Let \( T^* = S^*[P] \equiv \alpha^*(S^*[P]) \) be the result of a preliminary constant detection algorithm. The semantics transformer propagates constants along commands appearing within traces:

\[
t^*[\sigma, T^*] = \{ \tau^*[\sigma, T^*] \mid \sigma \in T \}, \quad \tau^*[\sigma, T^*] = \lambda i \cdot \tau^*[\sigma, T^*], \quad \tau^*[\rho, \tau^*] = \langle \rho, \tau^*[c, T^*(\text{lab}[c])] \rangle.
\]

This relies on the following command specialization algorithm:

\[
t^*[L_1 : A \rightarrow L_2], \quad \tau^*[A, \rho^*] \quad \Rightarrow \quad L_1 : \tau^*[A, \rho^*] \rightarrow L_2;
\]

\[
\tau^*[\rho, \rho^*] = \text{let } \rho_A = R[A]((\rho^t)[(\chi \in \rho, \chi \in \psi)])) \text{ in } \rho_A.
\]

6.4 Semantic Correctness of the Semantic Constant Propagation Transformation

The first aspect of semantic correctness is to prove that the observational abstraction \( \tau^* \) of the subject and transformed partial trace semantics are equal. This is trivial since environments are left untouched in the transformation.

6.5 Performance Correctness of the Semantic Constant Propagation Transformation

For performance correctness, the length of maximal traces is left unchanged while the evaluation of the transformed actions takes fewer elementary steps. Formally, the length of a finite trace is:

\[
\begin{align*}
|\sigma| & \triangleq \sum_{i=0}^{n-1} |\sigma_i|, \\
|\sigma| & \triangleq 1, \\
|\sigma| & \triangleq 0, \\
|\sigma| & \triangleq 1 + |\sigma| + |\sigma|, \\
|\sigma| & \triangleq 1 + |\sigma| + |\sigma|.
\end{align*}
\]

The weight of a set of traces is a mapping of trace observations to the maximal weight of the concrete traces with such observation:

\[
|T| = \lambda \zeta \in \alpha^*_{c}(T) \cdot \max\{|\sigma| \mid \alpha^*_{c}(\sigma) = \zeta\}.
\]

This is an abstraction:

\[
\text{po}(D' \subseteq C) \triangleq \sum_{n=0}^{\infty} \text{po}(D' \rightarrow N'; \leq)
\]

where \( \leq \) is the pointwise extension of \( \leq \). The performance correctness of semantic constant propagation follows from \( |T'| \leq |T| \).

6.6 Offline Syntactic Constant Propagation

The constant propagation algorithm \( t^*[P, T^*] \) is finally derived from the partial trace semantics \( F^*[P, T^*] \) by the abstraction \( \lambda T \cdot p^* \circ t^*[T, T^*] \), (where \( T^* = S^*[P] \) is the constant detection algorithm) using the fixpoint approximation theorem (1): \( p^* \circ t^*[F^*[P, T^*]] \equiv t^*[F^*[P, T^*]] \). where

\[
F^*[P, X] \triangleq \{ t^*[C, T^*(\text{lab}[C])] \mid C \in P \land \text{lab}[C] \in C[P] \} \cup \{ t^*[C', T^*(\text{lab}[C'])] \mid C' \in X : \text{act}[C] \neq \text{false} \land \text{lab}[C'] = \text{lab}[C'] \land C' \in P \}.
\]

As is classical in abstract interpretation [6], \( F^*[P, T^*] \) is formally derived from \( F_{\text{fix}}^*[P, T^*] \) by the commutation condition \( p^* \circ t^*[F_{\text{fix}}^*[P, T^*]] \equiv t^*[F_{\text{fix}}^*[P, T^*]] \). In practice, \( L \) is not computable so we compute \( lfp \equiv F^*[P, T^*] \) whence \( p^* \circ t^*[lfp \equiv F^*[P, T^*]] \equiv lfp \equiv F^*[P, T^*] \). Since the subject program is finite \( lfp \equiv F^*[P, T^*] \) immediately leads to an iterative constant propagation algorithm (where dead code is also partially eliminated).

6.7 Correctness of Offline Syntactic Constant Propagation

Finally, (9) implies that the semantics of program actions is unchanged by constant propagation which implies \( \alpha^*_{c}(S'[t^*[P, T^*]]) = \alpha^*_{c}(t^*[S'[P], T^*]) \) whence the correctness of syntactic constant propagation.

7. EXAMPLE 2: PARTIAL EVALUATION

It is now shown that online partial evaluation and binding time analysis based offline partial evaluation [1, 13, 14, 15] can be captured in the program transformation framework introduced in Sec. 3. This is applied to the imperative language of Sec. 4 which is similar to the one considered in [16, Ch. 4]. It is shown that the widening operation [4, 7] can be used in practice to enforce termination of the transformation expressed in fixpoint form which leads to a terminating iterative algorithm. Moreover widenings offer a continuum between online and offline partial evaluation as required in mixline partial evaluation [16, p. 153].

7.1 Online Partial Evaluation

Applying the framework of Sec. 3.9, we understand partial evaluation of a subject program (a syntactic transformation) as an abstract interpretation of a partial evaluation of its semantics (a syntactic transformation which is itself an abstraction of the subject semantics). Following [16, p. 78], “the idea of program point specialization is to incorporate the values of the static variables into the control point” so the set \( L \) of program labels is assumed to be of the form:

\[
L \triangleq \mathbb{N} \times \mathcal{C}
\]

where \( \mathbb{N} \) is the set of naturals and \( \mathcal{C} \) is the set of environments. We assume that all labels in the subject program belong to \( \mathbb{N} \times \{a\} \) (for short to \( \mathbb{N} \) up to an isomorphism).

7.1.1 Semantic Online Partial Evaluation

We define the environment and the label of the command of the first state in the non-empty trace \( \sigma \) respectively as \( \text{env}[\sigma] \triangleq \text{env}[\sigma_0] \) where \( \text{env}[\sigma_0, \rho_0] \triangleq \rho_0 \) and \( \text{lab}[\sigma] \triangleq \text{lab}[\sigma_0] \) where \( \text{lab}[\sigma_0, \rho_0] \triangleq \text{lab}[\rho_0] \). Similarly \( \alpha_{\text{act}}[\sigma] \triangleq \text{act}[\sigma] \) and \( \alpha_{\text{lab}}[\sigma] \triangleq \text{lab}[\sigma] \).

The partial evaluation of a set of non-empty traces \( T \) is the partial evaluation of the traces \( S \) in that set starting at a given program label \( L_0 \) for a given static environment \( \rho_0 \):

\[
\alpha_{\text{act}}[T](L_0, \rho_0) \triangleq \{ \text{PE}_{\text{act}}[\alpha_{\text{act}}(L_0, \rho_0), \sigma] \mid \sigma \in T \land \text{lab}[\sigma] = L_0 \land \rho_0 \in \text{env}[\sigma] \}.
\]

Observe that the semantic online partial evaluation is a functional abstraction:

\[\text{For short, this computation is not shown here. A similar one is detailed in Section 7.1.4.}\]
\[ \text{po}(\rho(\mathcal{D}^*); \subseteq) \xrightarrow{\text{def.} \mathcal{E} \times \mathcal{E}} \text{po}(L \times \mathcal{E}) \longrightarrow \rho(\mathcal{D}^*); \subseteq) \] (10)

The partial evaluation of a non-empty trace \( \sigma \) starting with an environment \( \rho \) specifies the residual/specialized computation when knowing the given static values \( \rho_0 \in \rho \) (we write \( \langle L_1, \rho_0 \rangle \triangleq \) if \( L_1 = 1 \) then \( 1 \) else \( \langle L_1, \rho_0 \rangle \)):

\[ \text{PE}_{\text{in}}([\rho, \quad \text{let } \langle \rho, \quad \text{let } L_1 = \{ L_1, \rho_0 \} \] in \n
\[ \langle \rho \setminus \text{dom} [\rho_0], L_0 = \mathcal{A} \rightarrow L_1^*; \text{PE}_{\text{in}}[\sigma][L_1, \rho_0] \rangle \) .

We define \( \text{PE}_{\text{in}}[\sigma] \langle L_0, \rho_0 \rangle \triangleq \) to handle stop commands and \( \text{PE}_{\text{in}}[\sigma] \langle L_0, \rho_0 \rangle \triangleq \) to cover the case of the empty trace.

7.1.2 Observational Abstraction

The observational abstraction of a set \( T \subseteq \mathcal{D}^* \) of traces gets rid of those traces in \( T \) not starting at the given program label \( L_0 \) with the given environment \( \rho_0 \):\n
\[ \alpha_{\text{PE}}^\text{obs}[T][L_0, \rho_0] \triangleq \{ \alpha_{\text{PE}}^\text{obs}[\sigma](\rho_0) \mid \sigma \in T \land \sigma \neq \} \Rightarrow \text{l \in \text{dom}[\rho_0]} \} \cdot \alpha_{\text{PE}}^\text{obs}[\sigma](\rho_0) \) .

By defining the concretization:

\[ \gamma_{\text{PE}}^\text{obs}[\mathcal{R}][L_0, \rho_0] \triangleq \{ \sigma \in \mathcal{D}^* \mid \sigma = \lor (\text{lab}[\sigma] = L_0 \land \rho_0 \in \text{env}[\sigma]) \Rightarrow \alpha_{\text{PE}}^\text{obs}[\sigma](\rho_0) \in \mathcal{R} \}, \]

we have the following observational abstraction (\( \mathcal{R} \) is the set of finite sequences of environments):

\[ \text{po}(\rho \setminus \text{dom} \rho_0); \subseteq) \xrightarrow{\text{def.} \mathcal{E} \times \mathcal{E}} \text{po}(L \times \mathcal{E}) \longrightarrow \rho(\mathcal{R}^*); \subseteq) . \]

7.1.3 Semantic and Performance Correctness of the Semantic Transformation

The semantic correctness of the semantic partial evaluation follows from the fact that the subject and specialized semantics have the same observed environments up to static variables:

\[ \alpha_{\text{obs}}^\text{obs}([\sigma] \langle L_0, \rho_0 \rangle) = \alpha_{\text{obs}}^\text{obs}[\alpha_{\text{obs}}^\text{obs}([T][L_0, \rho_0]) \langle L_0, \rho_0 \rangle] \]

The performance correctness of partial evaluation can be expressed by the performance abstraction introduced in Sec. 6.5 in that \( \alpha_{\text{obs}}^\text{obs}([T][L_0, \rho_0]) \subseteq \langle T \rangle \).\n
7.1.4 Fixpoint Online Partial Evaluation Semantics

We now compute the abstraction by the online partial evaluation abstraction \( \alpha_{\text{obs}}^\text{obs}([T][L_0, \rho_0]) \) defined in Sec. 7.1.1 of the partial trace semantics \( S'[P] \) expressed in the fixpoint form \( \text{fp}^* F'[P] \) of Sec. 4.5. We first establish the local commutation property necessary for fixpoint transfer (1).
Syntactic Source-to-Source Online Partial Eval-

By (10), the above local commutation property

Grouping the two terms together, we have:

by defining (again, to avoid a particular case for \texttt{stop} commands we assume, for short, that \(T(I, \rho_r) = \cdot \))

\[
F_{\text{PE}}[\rho] \in ((L \times \mathcal{E}) \rightarrow p(D^*)) \quad \text{and fixpoint transfer (1)}, \text{we conclude that} \quad F_{\text{PE}}[\rho] = F_{\text{PE}}[\rho^*] \quad \text{is correct}.
\]

7.1.5 Syntactic Source-to-Source Online Partial Evaluation

We now apply the semantics-to-syntax abstraction of Sec. 4.6 extended to

\[
\text{po}((L \times \mathcal{E}) \rightarrow p(D^*); \mathcal{C}) \xrightarrow{\mathcal{S}} \text{po}((L \times \mathcal{E}) \rightarrow \mathbb{P}; \mathcal{E}) \quad (12)
\]

where \(\mathcal{S}^*[T](\rho_0, \rho_1) \Rightarrow \rho^*[T](\rho_0, \rho_1)\) and \(\rho^*\) collects commands along a set of partial traces as defined in Sec. 4.6. The local semi-commutation property is computed for \(\subseteq\) as an approximation to \(\text{po}\) (and similarly for fixpoints as in Sec. 6.6):

\[
\mathcal{S}^*[T](\rho_0, \rho_1) \Rightarrow \rho^*[T](\rho_0, \rho_1)
\]

where \(p^*[T](\rho_0, \rho_1) \Rightarrow p^*[T](\rho_0, \rho_1)\) and \(p^*\) collects commands along a set of partial traces as defined in Sec. 4.6. The local semi-commutation property is computed for \(\subseteq\) as an approximation to \(\text{po}\) (and similarly for fixpoints as in Sec. 6.6):

\[
\mathcal{S}^*[T](\rho_0, \rho_1) \Rightarrow \rho^*[T](\rho_0, \rho_1)
\]

7.1.6 Online Partial Evaluation Semi-Algorith

The definition of the source-to-source partial evaluation in Sec. 7.1.5 is functional in that \(\text{ifp} \subseteq F_{\text{PE}}[\rho] \in ((L \times \mathcal{E}) \rightarrow \mathbb{P})\), in practice, partial evaluation algorithms consider a given specific value \(\langle L_0, \rho_0 \rangle\) of the initial argument only. The semi-algorithm to compute \(\text{ifp} \subseteq F_{\text{PE}}[\rho] \in ((L \times \mathcal{E}) \rightarrow \mathbb{P})\) follows from the first-order chaotic iterations of [5] popularized as minimal function graph [17]. We obtain the semi-algorithm of [16, Fig. 4.6, p. 80] using a working list \(\text{WL}\):

\[
\text{specialization}(\rho, L_0, \rho_0) =
\]

\[
\text{while} \quad \text{WL} \quad \text{contains an unmarked } (L, \rho) \quad \text{do}
\]

\[
\text{for all } L: A \rightarrow L_1; \quad E \quad \text{do}
\]

\[
\text{for all } L: A \rightarrow L_1; \quad E \quad \text{do}
\]

\[
\text{end end end;}
\]

\[
\text{return } \mathcal{Q}.
\]

Figure 8: A simple online specialization algorithm

7.2 Online Partial Evaluation with Widening

specialization is a semi-algorithm since it operates on an infinite complete lattice. As suggested in [5, 7], we can enforce convergence using widenings \(\vee\) and compute:

\[
\text{ifp} \subseteq \lambda T \cdot \lambda (L, \rho) \cdot T(L, \rho) \quad \text{fixpoint}
\]

The first widening \(\vee_1\) is to avoid an infinite iteration for the function body called with a given parameter (which is useless here since there are finitely many \((L, \rho), L \in \text{lab}[P]\) for a given \(\rho\)) and the second widening \(\vee_2\) is used to avoid infinitely many calls of the function with different parameters
Formally, a widening $\mathcal{V}_2$ is available for the program $P$, that is for all $L_0 \in \mathbb{L}$ and all $X_0 \subseteq X$, $S^* [P] \subseteq \mathcal{V}^{\mathcal{BTA}} (L_0, X_0)(\mathcal{V}[P])$. In order to avoid a particular case for stop commands, we extend $\delta \in \mathbb{L} \longmapsto \rho(X)$ to $\delta \in L \longmapsto \rho(X)$ by $\delta[i] \equiv \emptyset$.

### 7.4 Offline Partial Evaluation
Offline partial evaluation [16, Ch. 7] uses a division $\delta$ specifying which inputs are static and initial values $\rho_0|i|_{\delta,0}$ of the static variables at the program entry point $L_0$. It considers only those execution traces $\sigma$ starting at $L_0$ with values $\rho_i[\sigma]$ of the variables which are those assumed for the initial static variables (as specified by the condition $\rho_0|i|_{\delta,0} \subseteq \rho_i[\sigma]$):

$$\alpha_i^\delta(\delta, \sigma) = \{ \sigma \in T \wedge (\sigma \neq ) \Rightarrow (\rho_i[\sigma] = L_0 \wedge \rho_0|i|_{\delta,0} \subseteq \rho_i[\sigma]) \}$$

The partial evaluation of a trace computes static values and specializes actions for dynamic values as in the case of online partial evaluation (11) but for the fact that the considered static values are only those specified by the division:

$$\mathcal{P}^\text{E} \sigma(\delta, \rho_0 : A \rightarrow L_1; \mathcal{S}) \sigma(L_0, \rho_0) \equiv (13)$$

$\rho_i = \rho_0|i|_{\delta,0}$ and $L_0 = (L_0, \rho_0)$ and $(\rho_0, A_i) = \mathcal{R}[A]\rho_0'$ and $\rho_i' = \rho_i|_{\delta,1}|(L_0'$).

(Note that by induction on the length of traces and correctness of the BTA, we have $\delta(L_0) \subseteq \text{dom}(\rho_0)$ and $\delta(L_1) \subseteq \text{dom}(\rho_0)$.) The design of the offline specialization algorithm is then similar to that of online specialization as shown in Sec. 7.1.1. The resulting algorithm is therefore similar to Fig. 8 but for the fact that static environments (i.e. $\rho_0$, $\rho$ and $\rho_i$) are restricted to the division specified by the preliminary BTA (i.e. respectively $\rho_0|i|_{\delta,0}$, $\rho_0|i|_{\delta,1}$ and $\rho_0|i|_{\delta,L_1}$).

### 7.5 Mixline Partial Evaluation
Widensions offer a continuum between online and offline partial evaluation to achieve mixline partial evaluation. A simple example would consist in using the online partial evaulator of Fig. 8 with a widening, as considered in Sec. 7.2, but with a history-based one that computes for each label $L_1$ a division $\delta[L_1]$ which is initially $\mathcal{V}[P]$ and is progressively restricted during the partial evaluation by cumulating the intersection of the domains of all the static environments $\rho_i$ computed for that label $L_1$. The specialization $\mathcal{R}[A]$ would be restricted to that division $\delta[L_1]$ as in the offline partial evaluation of Sec. 7.4. With this widening, a BTA is performed during the mixline partial evaluation.

### 8. EXAMPLE 3: STATIC PROGRAM MONITORING
Program monitoring consists in restricting the possible executions $S^* [P]$ of a program $P \in \mathbb{P}$ in order to enforce a safety property. An example is the run-time checks for checking errors such as division by zero and out of bound array indexing. Another example is the enforcement of security policies by modifying object code for a target system before that system is executed so as to halt that system whenever it is about to violate some security policy of concern [10, 25]. This is similar to the idea of observers...
in synchronous programming, i.e. a program that observes the behavior of the subject program and decides whether it is correct [11].

8.1 Correctness of the Monitoring Transformation

Let $\mathcal{M}$ be the abstract specification of the program property to be enforced. The semantics of the transformed program $t_i^w[P]$ should, up to an observational abstraction $\alpha_{\mathcal{O}}$, be a subset of the executions of $P$ (so that $t_i^w[P]$ refines $P$) and satisfy the abstract specification $\mathcal{M}$. Formally, $\alpha_{\mathcal{O}}(S|[t_i^w[P]]) \subseteq \alpha_{\mathcal{O}}(S|[P]) \cap \alpha_{\mathcal{O}}(\gamma(M))$ (where equality is preferred since most precise and $\gamma(M)$ is the semantic meaning of the abstract specification $\mathcal{M}$ as defined below).

8.2 Observational Abstraction

We choose the abstraction $\alpha_{\mathcal{O}}$ to only observe the modifications to the successive environments during execution of $P$ that is $\alpha_{\mathcal{O}}(T) \triangleq (\alpha_{\mathcal{O}}(\sigma) \mid \sigma \in T)$ and for traces $\alpha_{\mathcal{O}}(T)$ $\triangleq \alpha_{\mathcal{O}}((p, c, \sigma)) \triangleq \alpha_{\mathcal{O}}((p, c')\sigma)$ and $\alpha_{\mathcal{O}}((p, c', \sigma)) \triangleq \rho_{var[p]} \alpha_{\mathcal{O}}((p', c')\sigma) \text{ if } \rho \neq \rho'$.

8.3 Reference Monitor

We choose the abstract monitoring specification $\mathcal{M}$ to be provided as a program $M \in \mathcal{P}$ called the reference monitor (as usual in the particular context of security policy enforcement [10, 25]), $\gamma(M)$ being now its semantics $S_i[M]$. (Any other mean for specifying finite sequences, such as automata [10, 25], grammars, safety temporal formula, etc. would do as well for $\mathcal{M}$ and could be handled equally well in the framework provided their semantics can be expressed in fixpoint form.) The subject program $P$ is assumed to have its labels in $L_P$ and its actions in $A_P$ so its execution traces belong to $\mathcal{D}^* [L_P \times A_P]$ where $\mathcal{D}^* [L, A]$ is the set of finite partial execution traces with labels in $L$ and actions in $A$. The reference monitor $\mathcal{M}$ is assumed to have its labels in $L_M$ and its actions in $A_M$ so its execution traces belong to $\mathcal{D}^* [L_M \times A_M]$. Actions in $A_M$ are specific to the monitor while actions in $A_P$ specify which actions of $P$ are allowed (in practice one would prefer abstract actions in $\mathcal{M}$ designating a set of concrete actions in $P$ such as $\neg A$ standing for all actions in $P$ but $A$).

8.4 Semantic Monitoring Transformation

The transformed program $t_i^w[P]$, which incorporates both the subject program $P$ and the reference monitor $\mathcal{M}$ has its labels in $L_P \times L_M$ (so $L = L_P \cup L_M \cup (L_P \times L_M)$) and its actions in $A = A_P \cup A_M$. A state $(\rho, (L, L')) : A \rightarrow (L, L')$ in a transformed trace must have $L$ when $A \not\in A_M$ (in which case the monitor $\mathcal{M}$ makes a step and the subject program $P$ makes no progress). To avoid a particular case for stop commands, we set $I \equiv (l, L) \equiv (L, l)$. The set of such partial execution traces is written $\mathcal{D}^* [L_P \times L_M, A]$. Intuitively, $t_i^w[P]$ is equivalent to $P$ but blocks when stopped by the reference monitor $\mathcal{M}$.

The projection of a trace $\sigma \in \mathcal{D}^* [L_P \times L_M, A]$ according to $P$ and $\mathcal{M}$ is defined as (starting with the projection of states):

$$\langle \rho, (L, L') : A \rightarrow (L, L') ; \rangle_\mathcal{P} \triangleq \langle \rho_{var[p]}, L: A \rightarrow L ; \rangle,$$

if $A \in A_P$;

$$\langle \rho, (L, L') : A \rightarrow (L, L') ; \rangle_{\mathcal{M}} \triangleq \langle \rho_{var[p]}, L': A \rightarrow L' ; \rangle;$$

$$\langle \sigma \rangle_\mathcal{P} \triangleq \langle \sigma \rangle_{var[p]}, (\sigma \rangle_{P}), \text{ if } act[\sigma] \in A_P;$$

$$\langle \sigma \rangle_{\mathcal{M}} \triangleq \langle \sigma \rangle_{var[p]}, (\sigma \rangle_{\mathcal{M}}), \text{ if } act[\sigma] \not\in A_P;$$

The program monitoring semantic transformation can now be defined as $t^w_i [T_P, T_M] \triangleq (\sigma \in D^* [L_P \times L_M, A] \mid \sigma_{\mathcal{P}} \in T_P \land \sigma_{\mathcal{M}} \in T_M)$. This is an abstraction $po(D^* [L_P \times L_M, A] ; \leq) \xhookrightarrow{\gamma_{\mathcal{M}}} po(D^* [L_P \times L_M, A] ; \leq)$. Its correctness follows from the fact that $\alpha_{\mathcal{O}}(t^w_i [S_i [P], S_i [M]]) = \alpha_{\mathcal{O}}(S_i [P]) \cap \alpha_{\mathcal{O}}(S_i [M])$.

8.5 Monitored Fixpoint Semantics

The monitored semantics $t^w_i [S_i [P], S_i [M]]$ can now be expressed in fixpoint form. In order to use (1), the pair of semantics of $P$ and $Q$ is first expressed in fixpoint form as $fp \xhookrightarrow{\delta} \lambda (T, T') \cdot (F_i[P], F_i[M], T')$. Then it is abstracted by the monitoring semantic transformation $t^w_i$ as $fp \xhookrightarrow{\delta} \lambda (T, T') \cdot (F_i[P], F_i[M], T')$. Thus, the algorithm is simpler than the monitoring($P, \mathcal{L}[P], M, \mathcal{L}[M]$) =

$$\text{while } \text{WH} \text{ contains an unmarked pair } (L, L') \not\equiv 1 \text{ do mark } (L, L') \text{ in } \text{WH};$$

for all $L: A \rightarrow L \in P$ and $L': A' \rightarrow L' \in M$ do

if $A = A'$ then step $k$ in source and monitor

$Q := Q \cup \{ (L, L') : A \rightarrow (L, L') \};$

WH := WH \cup \{ (L, L') \}$

elsif $A' \not\in A_M$ then control step $k'$ in monitor

$Q := Q \cup \{ (L, L') : A' \rightarrow (L, L') \};$

WH := WH \cup \{ (L, L') \}$

end end end;

return $Q$.

Figure 9: A simple program monitoring algorithm

8.7 Program Proof by Transformation

Finally, program monitoring can be used as a proof method. If the semantics of the transformed program $t_i^w[P]$ (maybe after static analysis and optimization) is empty (e.g. when
Theorem based model-checking \[27\]. When the observational abstraction of the semantics of the subject and transformed programs \(P\) and \(t^\omega[P]\) are equal (e.g. when, maybe after further optimizations such as partial evaluation, \(P\) and \(t^\omega[P]\) are isomorphic up to label renaming), then \(P\) does satisfy the specification \(M\). An example is the contrapositive automata-theoretic based model-checking \[27\].

This method of proving program properties by program transformation remains to be fully explored.

9. CONCLUSION

We have shown that program transformation can be formalized within abstract interpretation theory. This leads to a new construction of program transformations as syntactic approximations of provably correct semantic transformations. This has been applied to the simple case of constant propagation, to online and offline partial evaluation which is certainly the most widely applicable and practical classical program transformation and to program monitoring (such as security policy enforcement). The framework unifies the static analysis and transformation of programs within solid semantic foundations. For offline transformation we use a specification of the static analysis algorithm as an abstraction of the program semantics thus making the transformation correctness proof independent of the particular static program analysis algorithm which is used. By using widening techniques as well as abstract domain combination techniques this leads to formal methods for combining static analyses and transformations more intimately. Although illustrated on a low level imperative language, the formalization is language independent. It is indeed applicable to any computational process, for example to database search. Our formalization of program transformation by abstract interpretation makes very few hypotheses on the considered transformations and programming languages (only required to have some well-defined operational semantics) so that the model should be of very general scope.


10. REFERENCES