Observational Abstract Interpreters

RAVI MANGAL, Georgia Institute of Technology

We are interested in algorithmically finding proofs of program judgments. Some common strategies used by program proof search algorithms include computing semantic program invariants, making hypotheses about program behavior via dynamic checks embedded in the program, and using observations about program behavior to assist in proof search. We present *observational abstract interpreters*, a new meta-theoretic construction, that brings these three strategies for program reasoning together into a single approach. Similar to abstract interpreters, observational abstract interpreters compute semantic program invariants. However, in observational abstract interpreters, the invariants are permitted to be hypothetical, with program safety ensured via dynamic checks and hypotheses are made based on observed behavior of the program.

We formalize our ideas in the context of a simple higher-order language (λ_S). We develop a generic observational abstract interpreter for λ_S, drawing inspiration from the abstracting abstract machines (AAM) recipe for abstract interpreter construction. Observational abstract interpreters have a monadic structure and are capable of making hypotheses based on program observations during the computation of semantic program invariants. We present an instantiation of the generic observational abstract interpreter for λ_S, yielding an observational variant of interval analysis for λ_S.

CCS Concepts: • **Software and its engineering → General programming languages**;

Additional Key Words and Phrases: abstract interpreters, dynamic checks, observations

ACM Reference Format:
Ravi Mangal. 2020. Observational Abstract Interpreters. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2020), 28 pages.

1 INTRODUCTION

Program verification, as used colloquially, refers to the practice of algorithmically finding program proofs, i.e., proofs of program judgments. These program judgments come in many forms, common forms are either type-theoretic judgments like $\Gamma \vdash e : t$ saying that in context Γ program e has type t, or program logic judgments of the form, $\{P\}e\{Q\}$, particularly when e is from an effectful language, where P is a pre-condition and Q is a post-condition of e.[1] Irrespective of the form of the judgment, a common step in the proof strategy employed by proof search algorithms is to compute semantic invariants of e which are then further used to construct the proofs of program judgments. The use of semantic invariants is particularly common when the programs or terms e are only partially annotated or are completely unannotated (à la Curry where programs are thought to be terms from an untyped language and the type system is extrinsic [Reynolds 2000]). Informally, a semantic invariant is a simplified representation of the meaning of a program and practically, one wants these representations to be efficiently computed even when the program under analysis is non-terminating. A unifying perspective on algorithms for computing such invariants is provided by the theory of abstract interpretation [Cousot and Cousot 1977, 1992].

In general, the decision problems addressed by program verification are undecidable [Rice 1953]. Even in the instances where the problems are decidable, the ability of an invariant-based proof search algorithm to find a proof (or a counterexample) crucially depends on the computed invariants. Invariants computed by abstract interpreters, in turn, depend on the abstract semantic domain and

[1] There are many connections between these two judgment forms that we do not elaborate here.

Author's address: Ravi Mangal, Georgia Institute of Technology, rmangal3@gatech.edu.

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

the abstract semantic function used to construct the abstract interpreter. The theory of abstract interpretation defines language semantics as a pair of a concrete semantic domain and a semantic function. The theory also defines the manner in which the concrete semantics should relate to an abstract semantics so that the invariants computed using the abstract semantics can be soundly used in the background type theory/program logic for constructing a program proof. However, defining an abstract semantics that leads to efficient computation of useful invariants requires creativity and theoretical expertise.

Many ideas have been presented in the literature for making the process of designing abstract semantics "easier" - Van Horn and Might [2010] present a systematic approach for constructing an abstract interpreter starting from abstract machine semantics of higher-order languages and a number of follow-on works extend these ideas [Darais et al. 2017, 2015; Keidel and Erdweg 2019; Keidel et al. 2018; Sergey et al. 2013]; calculational abstract interpretation yields the abstract semantic function automatically given the concrete semantics and the abstract semantic domain [Cousot 1999; Darais and Horn 19ed; Reps et al. 2004; Thakur et al. 2015]; in the counterexample-guided abstraction refinement (CEGAR) style of abstract interpretation [Clarke et al. 2003], the designer defines a set (finite or infinite) of "correct" abstract semantics and, given a specific program judgment, the CEGAR algorithm searches through this set for an abstract semantics that can efficiently yield a proof (or a counterexample) of the judgment. While all these ideas have helped make the design of effective abstract interpreters easier, the design process still involves much human ingenuity.

A different, increasingly common, proof strategy employed by proof search algorithms is to modify the program under study and embed it with run time or dynamic checks. This allows making hypotheses about program behavior such that a proof of the required program judgment can be constructed. This type of reasoning has been popularized by the gradual typing [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006] and hybrid typing [Flanagan 2006; Knowles and Flanagan 2010] philosophy as well as the work on using logical abduction for program reasoning [Calcagno et al. 2011; Dillig et al. 2012]. Ideally, we want to compute the weakest hypotheses that allow the construction of a program proof but inferring such hypotheses is not trivial.

We are interested in the design of proof search algorithms that combine the use of semantic invariants and dynamic checks. Apart from recent work on gradual liquid type inference [Vazou et al. 2018] and gradual program verification [Bader et al. 2018], such a combination has been relatively under explored formally. In this work, we present the design of a new class of abstract interpreters that compute semantic invariants while making hypotheses about program behavior, embedded as dynamic checks in the program. These hypotheses help the abstract interpreter compute potentially stronger semantic invariants, at the cost of the overheads of dynamic run time checks. A key challenge in such hypothesis-based reasoning is automatically computing the appropriate hypothesis. Typically, the computation of these hypotheses is guided by the proof goal. In our abstract interpreter design, we instead rely on observations about the program behavior to infer the hypotheses. Intuitively, the idea is to make hypotheses that are consistent with the observed behavior of the program. This observational style of reasoning motivates our use of the term, *observational abstract interpreters*, to refer to the class of abstract interpreters that we propose.

The benefit of an observational reasoning style, particularly in combination with the hypotheses-based reasoning, is that we no longer need to derive custom proof goal guided algorithms, specific to the type theory or program logic we are working with, for computing the appropriate hypotheses. More interestingly, such observational, hypothetical proofs, can be used to make judgments about the program behavior in the "commonly" observed ways of using the program, even if the same judgment cannot be proven for the program in general. On the other hand, an obvious drawback of using program observations (instead of the proof goal) for computing hypotheses is that the

$$
\begin{array}{rcl}
 & i \in \mathbb{Z} \quad x \in Var \quad l \in Lbl & \\
a \in Atom & ::= & i \mid x \mid \boldsymbol{\lambda}(x).e \mid \textbf{abort} \\
\oplus \in IOp & ::= & + \mid - \\
\odot \in Op & ::= & \oplus \mid @ \\
e \in Exp & ::= & (a)^l \mid (e \odot e)^l \mid (\textbf{if0}(e)\{e\}\{e\})^l
\end{array}
$$

Fig. 1. $\lambda_S(\lambda_{SA})$ language syntax

computed hypotheses are not guaranteed to be strong enough to allow the construction of a program proof. In any case, we believe that this combination of invariant-based reasoning with hypotheses-based reasoning, where the hypotheses are inferred from program observations is an interesting point worth further exploration.

We formalize our ideas in the context of a simple higher-order language (λ_S). In particular, starting from an abstract machine semantics of λ_S, we demonstrate the construction of a generic observational abstract interpreter for λ_S, and in the process, we formally define the notion of program observations as well as the notion of correctness or soundness for an observational abstract interpreter. Our formal development is heavily inspired by the abstracting abstract machines (AAM) [Van Horn and Might 2010] style of abstract interpreter construction. Observational abstract interpreters are structured as monadic abstract interpreters [Darais et al. 2017, 2015; Sergey et al. 2013] that reify the notion of an AAM-style interpreter. We believe that the recipe we present here for constructing observational abstract interpreters of λ_S can also be applied to other languages.

Our main contributions are as follows - (i) we propose observational abstract interpreters, a synthesis of invariant-based reasoning about programs with hypothesis-based reasoning and observational program reasoning, (ii) we formally construct a generic observational abstract interpreter for λ_S, a higher-order language, (iii) we present an instantiation of the generic observational abstract interpreter for λ_S, yielding an observational interval analysis for programs in λ_S.

2 LANGUAGE DEFINITION

We present our ideas with the help of λ_S, a higher-order language with built-in integers and conditionals. The language is fairly standard, and we adopt the syntax and semantics from [Darais et al. 2015]. λ_S syntax is defined in Figure 1. Note that function application is explicitly represented using the @ operator. Figure 1 also describes the syntax of λ_{SA}, which additionally allows programs with **abort** expressions (the gray background color is intended to highlight that **abort** expressions are only allowed in λ_{SA} programs, but not in λ_S programs). These **abort** expressions enable dynamic checks to be embedded in the programs. We distinguish between λ_S and λ_{SA} for ease of formal presentation. We design observational abstract interpreters that are capable of analyzing λ_S programs to produce hypothetical semantics invariants. These hypotheses are then embedded in the original λ_S program with the help of **abort** expressions, producing a λ_{SA} program. Note that every expression in a λ_S (λ_{SA}) is associated with a unique label, drawn from an infinitely large set of labels (*Lbl*). To avoid notational clutter, we do not show the labels in the rest of the paper, but assume that such a label always exists. Moreover, we assume the existence of a function get-Label that accepts an expression and returns the label associated with the expression.

The semantics of λ_S (and λ_{SA}) are presented in Figure 2. We define the language semantics using the formalism of abstract machines. Before describing the semantics, we make a note on the metalanguage used in Figure 2 and the rest of the paper. Our metalanguage notation resembles Haskell syntax, though we freely use other syntactic constructs. Function application is notated

$$
\begin{array}{rcl}
t \in Time & := & Exp^* \\
a \in Addr & := & Var \times Time \\
\rho \in Env & := & Var \rightharpoonup Addr \\
s \in Store & := & Addr \rightharpoonup Val \\
kf \in KFrame & := & Frame \times Env \\
ka \in KAddr & := & Time \\
ks \in KStore & := & KAddr \rightharpoonup KFrame \times KAddr \\
c \in Clo & ::= & \langle \boldsymbol{\lambda}(x).e, \rho \rangle \\
v \in Val & ::= & i \mid c \mid \textbf{abort} \\
fr \in Frame & ::= & \square \odot e \mid v \odot \square \mid \textbf{if0}(\square)\{e\}\{e\} \\
\sigma \in \Sigma & ::= & \langle e, \rho, s, ka, ks, t \rangle
\end{array}
$$

(a) Type definitions

$$
\begin{array}{rcl}
[\![\cdot]\!]_A & : & Atom \to (Env \times Store \rightharpoonup Val) \\
[\![i]\!]_A(\langle \rho, s \rangle) & := & i \\
[\![x]\!]_A(\langle \rho, s \rangle) & := & s(\rho(x)) \\
[\![\boldsymbol{\lambda}(x).e]\!]_A(\langle \rho, s \rangle) & := & \langle \boldsymbol{\lambda}(x).e, \rho \rangle \\
[\![\cdot]\!]_\delta & : & IOp \to (\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}) \\
[\![+]\!]_\delta(\langle i_1, i_2 \rangle) & := & i_1 + i_2 \\
[\![-]\!]_\delta(\langle i_1, i_2 \rangle) & := & i_1 - i_2
\end{array}
$$

(b) Denotational semantics of atomic expressions

$$
\begin{array}{rcll}
\cdot \rightsquigarrow \cdot & : & P(\Sigma \times \Sigma) & \\
\langle e_1 \odot e_2, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e_1, \rho, s, ka, ks', t' \rangle & \text{where} \\
t' & := & (e_1 \odot e_2) :: t & \\
ks' & := & ks[t' \mapsto \langle \langle \square \odot e_2, \rho \rangle, ka \rangle] & \\
\\
\langle \textbf{if0}(e_1)\{e_2\}\{e_3\}, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e_1, \rho, s, ka, ks', t' \rangle & \text{where} \\
t' & := & (\textbf{if0}(e_1)\{e_2\}\{e_3\}) :: t & \\
ks' & := & ks[t' \mapsto \langle \langle \textbf{if0}(\square)\{e_2\}\{e_3\}, \rho \rangle, ka \rangle] & \\
\\
\langle \textbf{abort}, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle \textbf{abort}, \rho, s, t, ks, t \rangle & \\
\\
\langle a, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e, \rho', s, t, ks', t' \rangle & \text{where} \\
t' & := & a :: t & \\
\langle \langle \square \odot e, \rho' \rangle, ka' \rangle & := & ks(ka) & \\
ks' & := & ks[t' \mapsto \langle \langle [\![a]\!]_A(\langle \rho, s \rangle) \odot \square, \rho \rangle, ka' \rangle] & \\
\\
\langle a, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e, \rho'', s', ka', ks, t' \rangle & \text{where} \\
t' & := & a :: t & \\
\langle \langle \langle \boldsymbol{\lambda}(x).e, \rho' \rangle @ \square, \rho' \rangle, ka' \rangle & := & ks(ka) & \\
\rho'' & := & \rho'[x \mapsto \langle x, t' \rangle] & \\
s' & := & s[\langle x, t' \rangle \mapsto [\![a]\!]_A(\langle \rho, s \rangle)] & \\
\\
\langle i_2, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle i, \rho, s, ka', ks, t' \rangle & \text{where} \\
t' & := & i_2 :: t & \\
\langle \langle i_1 \oplus \square, \rho' \rangle, ka' \rangle & := & ks(ka) & \\
i & := & [\![\oplus]\!]_\delta(\langle i_1, i_2 \rangle) & \\
\\
\langle i, \rho, s, ka, ks, t \rangle & \rightsquigarrow & \langle e, \rho', s, ka', ks, t' \rangle & \text{where} \\
t' & := & i :: t & \\
\langle \langle \textbf{if0}(\square)\{e_1\}\{e_2\}, \rho' \rangle, ka' \rangle & := & ks(ka) & \\
e & := & \textbf{if } i = 0 \textbf{ then } e_1 \textbf{ else } e_2 &
\end{array}
$$

(c) Abstract machine semantics

Fig. 2. $\lambda_S(\lambda_{SA})$ concrete semantics in the form of an abstract machine

$$
\begin{array}{rcl}
\overline{\text{init-States}} & : & Exp^{-} \to \mathcal{P}(\Sigma) \\
\overline{\text{init-States}}(e) & := & \textbf{let } \rho := \{\langle x, \langle x, \epsilon\rangle\rangle \mid x \in FV(e)\} \textbf{ in} \\
 & & \textbf{let } \text{init-Store} := \{\{\langle\langle x, \epsilon\rangle, v_x\rangle \mid x \in FV(e)\} \mid \textstyle\bigwedge_{x \in FV(e)} v_x \in \mathbb{Z}\} \textbf{ in} \\
 & & \{\langle e, \rho, s, \epsilon, \bot, \epsilon\rangle \mid s \in \text{init-Store}\}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\cdot]\!]_{CI} & : & Exp^{-} \to \mathcal{P}(\Sigma) \\
[\![e]\!]_{CI} & := & \textbf{lfp } \lambda(x).\ x \cup \overline{\text{init-States}}(e) \cup \{\sigma_2 \mid \sigma_1 \in x \wedge \sigma_1 \rightsquigarrow \sigma_2\}
\end{array}
$$

Fig. 3. $\lambda_S(\lambda_{SA})$ collecting semantics

as $f(e)$, where f is the function applied to e. Pairs and tuples are notated by $\langle\cdot\rangle$. We reserve = to explicitly notate equality, with := used to notate definitions, and ::= notates datatype definitions. Wherever necessary, we explain the notation that we use.

The abstract machine semantics of λ_S (and λ_{SA}) is defined as a transition relation ($\rightsquigarrow$) on the set Σ of abstract machine states. An abstract machine state is a 6-tuple consisting of a program/expression, an environment (*Env*), a store for values (*Store*), a store for continuations (*KStore*) that are linked together (similar to a call stack), the address of the next continuation (*KAddr*), and a time component (*Time*). The abstract machine semantics presented here is similar to the CESK machine [Felleisen and Friedman 1987], except that the continuations are threaded through the store, and the time component is used to compute a new address for allocation in the value or continuation store. As mentioned earlier, the abstract machine design presented here follows the design by Darais et al., which is itself based on work by Van Horn and Might [Van Horn and Might 2010]. Since values and continuations are both allocated in their respective stores, by restricting the number of distinct locations/addresses in the store, one can easily abstract the abstract machine, yielding an abstract interpreter for the language, an observation that first appeared in [Van Horn and Might 2010].

Figure 2a defines the different components of an abstract machine state. We would like to draw notice to the definition of *Time* and *Addr*. *Time* is defined as a sequence of expressions, while an address is a pair of a variable name and time. We assume that each of the type (or set) defined here has the structure of a lattice. The semantics of atomic expressions and primitive operations are defined denotationally (2b), and the abstract machine semantics for compound expressions are defined by a relation (2c). Note that if the abstract machine encounters an **abort** expression while executing a λ_{SA} program, it steps to an unmodified state.

Following all these definitions, we are finally ready to define the notion of "meaning" of a program, also referred to as *collecting semantics* in the abstract interpretation literature. Figure 3 defines the collecting semantics of $\lambda_S(\lambda_{SA})$. Note that the collecting semantics are not defined for all the expressions (*Exp*) in our language. Instead, we only consider programs where the free variables are of type $\mathbb{Z}$, and name this set of expressions, Exp^{-}. The meaning of a program/expression in Exp^{-} is described in terms of abstract machine states. Intuitively, the meaning of a program is the set of all abstract machines states that are "reachable" from a set of "initial" states. Let us unpack this definition. Given a program e, the definition of initial states ($\overline{\text{init-States}}$) in Figure 3 states that, if a program e has no free variables, then the set of initial states is just the singleton set, $\{\langle e, \bot, \bot, \epsilon, \bot, \epsilon\rangle\}$. For programs with free variables of type $\mathbb{Z}$ (set of free variables is represented by *FV*), the set of initial states is defined such that all possibles ways of "closing" the program, i.e., assigning values to the free variables, are represented in the set. In Figure 3, this is captured by the definition of init-Store, which uses the set-builder notation in a nested manner. Given a free variable x, we assume that the initial value assigned to x is stored at address $\langle x, \epsilon\rangle$ in the store s. Then, the collecting semantics, notated by $[\![\cdot]\!]_{CI}$, is defined as the least fixed point of a function of type $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. This function uses the definitions of $\overline{\text{init-States}}$ and the transition relation

$\rightsquigarrow$ describing our abstract machine semantics. Defining a collecting semantics for expressions with free variables of function type is a problem of independent interest, and by only considering programs from Exp^{-}, we avoid dealing with that issue in this paper.

3 MONADIC INTERPRETERS: CONCRETE AND ABSTRACT

The Van Horn-Might [Van Horn and Might 2010] style of abstract machine semantics for higher-order languages makes it easy to refactor the abstract machine such that designing an abstract interpreter simply becomes a matter of redefining some interfaces (expressible as type classes in Haskell or modules in ML). Sergey et al. first noticed that the Van Horn-Might abstract machine can be refactored using monads. That interpreters for higher-order languages can be modularized and structured monadically has been known for a while [Liang et al. 1995; Wadler 1992], but using the monadic structure to ease the design of abstract interpreters and simplify their proofs of correctness has only been recently investigated [Darais et al. 2017, 2015; Keidel and Erdweg 2019; Keidel et al. 2018; Sergey et al. 2013]. These recent advances play an important role in our design of observational abstract interpreters. In this section, we describe how the abstract machine semantics for λ_S can be modularized and expressed monadically, closely following the work of Darais et al. We also show the manner in which the resulting monad can be instantiated to yield semantics equivalent to the collecting semantics defined in Figure 3, as well as an abstract interpreter for an interval analysis of λ_S programs.

Figure 4 describes the design of a generic monadic interpreter for programs in λ_S with free variables of type $\mathbb{Z}$. The design of the monadic interpreter is based on the intuition that the computation performed by the interpreter (or the abstract machine) primarily depends on the structure of the expression being interpreted, and the interaction with the other components of the abstract machine state, like the environment and the store, can be hidden behind a monadic interface. This monadic interface in defined in Figure 4a. Our metalanguage supports Haskell-like typeclasses [Wadler and Blott 1989], and we define a typeclass $\mathring{m}$ that includes standard monadic operations like bind and return. In addition, the monad is required to support a number of get and put operations for interacting with the abstract machine state components. Additionally, the monad is also required to support non-deterministic choice operation $\langle + \rangle$. Besides the monad typeclass, the monadic interpreter design also requires abstracting other types that the interpreter interacts with via corresponding typeclasses. In our notation, we distinguish typeclass names from type names by using a small circle ($\mathring{name}$) over the typeclass names. The typeclass $\mathring{Time}$ has an associated operation, tick. The typeclass $\mathring{Val}$ has a number of operations associated with it that map from values of type $\mathbb{Z}$ and closures to elements of types instantiating $\mathring{Val}$, and vice versa. More details about these operations can be read in Section 4.2 of [Darais et al. 2015]. Finally, we expect $\mathring{Time}, \mathring{Val}, \mathring{Addr}, \mathring{Env}, \mathring{Store}, \mathring{KFrame}, \mathring{KAddr}, \mathring{KStore}$, and $\mathring{\Sigma}$ to all have a lattice structure, i.e., they support the lattice operations $\sqcup$, $\sqcap$, and $\sqsubseteq$, as well as define lattice elements $\top$ and $\bot$.

Figure 4b defines the step^{m} function describing a single step of the monadic interpreter. First, a comment on notation - we use the do notation from Haskell as well as ; for sequencing monadic operations. So $x \leftarrow s_1; s_2$ is syntactic sugar for $\text{bind}(s_1)(\lambda(x).\ .s_2)$, while

$$\begin{array}{l}\textbf{do}\\ x \leftarrow s_1\\ s_2\end{array}$$

is syntactic sugar for $\text{bind}(s_1)(\lambda(x).\ s_2)$. Moreover, we allow combining these notations. The step^{m} function uses a number of helper functions, defined in Figure 4c. A further comment on notation - in order to check if a partial may, say ρ, is defined for a certain key, say x, we use the notational

$\underline{\mathring{Time}}$
tick : $Exp^- \times \mathring{Time} \to \mathring{Time}$

$\underline{\mathring{Val}}$
int-I : $\mathbb{Z} \to \mathring{Val}$
if0-E : $\mathring{Val} \to \mathcal{P}(Bool)$
clo-I : $\mathring{Clo} \to \mathring{Val}$
clo-E : $\mathring{Val} \to \mathcal{P}(\mathring{Clo})$
$[\![\oplus]\!]_{m\delta}$: $\mathring{Val} \times \mathring{Val} \to \mathring{Val}$

$\mathring{Addr} := Var \times \mathring{Time}$
$\mathring{Env} := Var \rightharpoonup \mathring{Addr}$
$\mathring{Clo} ::= \langle \boldsymbol{\lambda}(x).e, \rho \rangle$
$\mathring{Store} := \mathring{Addr} \rightharpoonup \mathring{Val}$
$\mathring{KFrame} := Frame \times \mathring{Env}$
$\mathring{KAddr} := \mathring{Time}$
$\mathring{KStore} := \mathring{KAddr} \rightharpoonup \mathcal{P}(\mathring{KFrame} \times \mathring{KAddr})$

$\underline{\mathring{\Sigma}}$
init-States : $Exp^- \to \mathring{\Sigma}$

$\underline{\mathring{m}}$
return : $\forall A.A \to \mathring{m}(A)$
bind : $\forall A, B.\mathring{m}(A) \to (A \to \mathring{m}(B)) \to \mathring{m}(B)$
get-Env : $\mathring{m}(\mathring{Env})$
put-Env : $\mathring{Env} \to \mathring{m}(1)$
get-Store : $\mathring{m}(\mathring{Store})$
put-Store : $\mathring{Store} \to \mathring{m}(1)$
get-KAddr : $\mathring{m}(\mathring{KAddr})$
put-KAddr : $\mathring{KAddr} \to \mathring{m}(1)$
get-KStore : $\mathring{m}(\mathring{KStore})$
put-KStore : $\mathring{KStore} \to \mathring{m}(1)$
get-Time : $\mathring{m}(\mathring{Time})$
put-Time : $\mathring{Time} \to \mathring{m}(1)$
mzero : $\forall A.\mathring{m}(A)$
$\cdot\langle + \rangle \cdot$: $\forall A.\mathring{m}(A) \times \mathring{m}(A) \to \mathring{m}(A)$

$\alpha^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$: $(\mathring{\Sigma} \to \mathring{\Sigma}) \to (Exp^- \to \mathring{m}(Exp^-))$
$\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$: $(Exp^- \to \mathring{m}(Exp^-)) \to (\mathring{\Sigma} \to \mathring{\Sigma})$

(a) Type definitions

Fig. 4. λ_S monadic interpreter

```
step^m : Exp^- → m̊(Exp^-)
step^m(e) := do
  ρ ← get-Env
  e'' ← case e of
    e1 ⊙ e2 → tick^m(e); push(⟨□ ⊙ e2, ρ⟩); return(e1)
    if0(e1){e2}{e3} → do
      tick^m(e); push(⟨if0(□){e2}{e3}, ρ⟩); return(e1)
    a → do
      v ← [[a]]_mA; fr ← pop
      case fr of
        ⟨□ ⊙ e', ρ'⟩ → do
          tick^m(e); put-Env(ρ'); push(⟨v ⊙ □, ρ⟩); return(e')
        ⟨v'@□, ρ'⟩ → do
          tick^m(e); t ← get-Time; s ← get-Store
          ⟨λ(x).e', ρ''⟩ ← ↑_p(clo-E(v'))
          put-Env(ρ''[x ↦ ⟨x, t⟩])
          put-Store(s ⊔ [⟨x, t⟩ ↦ v]); return(e')
        ⟨v' ⊕ □, ρ'⟩ → tick^m(e); return([[⊕]]_mδ(⟨v', v⟩))
        ⟨if0(□){e1}{e2}, ρ'⟩ → do
          tick^m(e); put-Env(ρ'); b ← ↑_p(if0-E(v)); refine(⟨a, b⟩)
          if (b) then return(e1) else return(e2)
        ⊥ → return(e)
  return(e'')
```

(b) Step function

```
[[·]]_mA : Atom → m̊(V̊al)
[[i]]_mA := return(int-I(i))
[[x]]_mA := do
  ρ ← get-Env; s ← get-Store
  if (x ∈ ρ) then return(s(ρ(x))) else return(⊥)
[[λ(x).e]]_mA := ρ ← get-Env; return(clo-I(⟨λ(x).e, ρ⟩))

push : KF̊rame → m̊(1)
push(fr) := do
  ka ← get-KAddr; ks ← get-KStore; ka' ← get-Time
  put-KStore(ks ⊔ [ka' ↦ ⟨fr, ka⟩]); put-KAddr(ka')

pop : m̊(KF̊rame)
pop := do
  ka ← get-KAddr; ks ← get-KStore;
  if (ka ∉ ks) then return(⊥)
  else ⟨fr, ka'⟩ ← ↑_p(ks(ka)); put-KAddr(ka'); return(fr)

↑_p : ∀A.P(A) → m̊(A)
↑_p({a1, ..., an}) := return(a1)⟨+⟩...⟨+⟩return(an)

refine : Atom × Bool → m̊(1)
refine(⟨i, b⟩) := return(1)
refine(⟨x, b⟩) := do
  ρ ← get-Env; s ← get-Store
  if (b) then put-Store(s[ρ(x) ↦ int-I(0)]) else return(1)

tick^m : Exp^- → m̊(1)
tick^m(e) := do
  t ← get-Time
  put-Time(tick(⟨e, t⟩))
```

(c) Helper functions

$$[\![\cdot]\!]_m \quad : \quad Exp^- \to \mathring{\Sigma}$$
$$[\![e]\!]_m \quad := \quad \mathbf{lfp}\ \lambda(x).\ x \sqcup \text{init-States}(e) \sqcup (\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}}(\text{step}^{\text{m}}))(x)$$

(d) Collecting semantics

Fig. 4. λ_S monadic interpreter

shortcut $x \in \rho$. The structure of the step$^{\mathsf{m}}$ function closely resembles the abstract machine transition relation defined in Figure 2c. The helper function $\uparrow_p$ helps hide the non-determinism behind the monadic interface. While the concrete interpreter for λ_S does not exhibit any non-determinism, we will soon see that the abstract interpreter is non-deterministic. Similarly, the function refine helps the abstract interpreter compute more precise results, particularly in cases where the branch taken by the conditional cannot be resolved.

Finally, Figure 4d defines the collecting semantics of a λ_S program in Exp^- using the monadic step$^{\mathsf{m}}$ function. Note that the type signature of step$^{\mathsf{m}}$ ($Exp^- \to \mathring{Exp}^-$) is incompatible with least-fixed point computation needed for computing the meaning of a program. As in [Darais et al. 2015], this problem is solved by defining a function $\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$ that maps the monadic step$^{\mathsf{m}}$ function, to a transition function of type $\mathring{\Sigma} \to \mathring{\Sigma}$, that can be iteratively invoked to compute the required least fixed point. The function $\alpha^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$ does the opposite, with $\alpha^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$ and $\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}}$ representing a Galois connection between $Exp^- \to \mathring{m}(Exp^-)$ and $\mathring{\Sigma} \to \mathring{\Sigma}$.

3.1 Concrete Monadic Interpreter

The monadic concrete interpreter for λ_S is derived by instantiating the typeclasses defined in Figure 4. These typeclass instantiations are described in Figure 5. We make sure that the monadic interpreter is instantiated such that the resulting "concrete" monadic collecting semantics (notated by $\overline{[\![\cdot]\!]_m}$) is equivalent to the collecting semantics ($[\![\cdot]\!]_{Cl}$) defined in Figure 3. Notationally, concrete typeclass instantiations are indicated by a horizontal line over the typeclass names (for instance, $\overline{Time}$).

Note that $\overline{Time}$, $\overline{Val}$, $\overline{Clo}$, $\overline{Addr}$, $\overline{Env}$, $\overline{Store}$, and $\overline{KAddr}$ reuse the corresponding definitions from Figure 2a for the standard abstract machine semantics of λ_S. However, $\overline{KStore}$, i.e., the continuation store, is defined such that every address is mapped to a set of continuations. However, these sets are always singleton in the concrete semantics. The meanings of programs are elements of set $\overline{\Sigma}$, defined as the powerset of the set of abstract machine states. The lattice operations for $\overline{\Sigma}$, defined in Figure 5b, are straightforward. In the collecting semantics, we reuse the definition of $\overline{\text{init-States}}$ from Figure 3 (ignoring the difference in the definitions of $\overline{KStore}$ and $KStore$ since it does not have any discernible effect on the definition of $\overline{\text{init-States}}$).

The correctness of the monadic concrete collecting semantics with respect to the standard collecting semantics of λ_S is formally stated by the following proposition.

Proposition 1. *(Equivalence of $[\![\cdot]\!]_{Cl}$ and $\overline{[\![\cdot]\!]_m}$)*
$\forall e \in Exp^-.\ [\![e]\!]_{Cl} = \overline{[\![e]\!]_m}$

A proof of this equivalence can be found in prior works ([Darais et al. 2015]), and since our definitions of the standard collecting semantics and the monadic semantics presented here closely follows that of Darais et al., we do not present the proof here.

3.2 Abstract Monadic Interpreter

The flexibility and modularity afforded by the monadic design of the λ_S interpreter can be appreciated as one sets out to design an abstract interpreter for the language. We present a monadic abstract interpreter for λ_S that is capable of performing interval analysis of λ_S programs. As with the monadic concrete interpreter, we only need to instantiate the typeclasses in order to yield the abstract interpreter. We notate typeclass instances for the abstract interpreter with a hat over the typeclass name (for instance, $\widehat{Time}$).

$t \in \overline{Time} := Time$
$\mathsf{tick}(\langle e, t\rangle) := e :: t$

$v \in \overline{Val} := Val$
$\mathsf{int\text{-}I}(i) := i$
$\mathsf{if0\text{-}E}(v) := \textbf{if } (v = 0) \textbf{ then true else false}$
$\mathsf{clo\text{-}I}(c) := c$
$\mathsf{clo\text{-}E}(v) := v$
$\overline{[\![+]\!]}_{m\delta}(\langle v, v'\rangle) := v + v'$
$\overline{[\![-]\!]}_{m\delta}(\langle v, v'\rangle) := v - v'$

$a \in \overline{Addr} := Addr$
$\rho \in \overline{Env} := Env$
$s \in \overline{Store} := Store$
$kf \in \overline{KFrame} := KFrame$
$ka \in \overline{KAddr} := KAddr$
$ks \in \overline{KStore} := \overline{KAddr} \rightharpoonup \mathcal{P}(\overline{KFrame} \times \overline{KAddr})$
$c \in \overline{Clo} := Clo$

$\psi \in \overline{\Psi} := \overline{Env} \times \overline{Store} \times \overline{KAddr} \times \overline{KStore} \times \overline{Time}$
$\overline{\sigma} \in \overline{\Sigma} = \mathcal{P}(Exp^{-} \times \overline{\Psi})$

$\alpha^{\overline{\Sigma} \leftrightarrow \overline{m}} : (\overline{\Sigma} \to \overline{\Sigma}) \to (Exp^{-} \to \overline{m}(Exp^{-}))$
$\alpha^{\overline{\Sigma} \leftrightarrow \overline{m}}(f)(e)(\psi) := f(\{\langle e, \psi\rangle\})$
$\gamma^{\overline{\Sigma} \leftrightarrow \overline{m}} : (Exp^{-} \to \overline{m}(Exp^{-})) \to (\overline{\Sigma} \to \overline{\Sigma})$
$\gamma^{\overline{\Sigma} \leftrightarrow \overline{m}}(f)(\overline{\sigma}) := \bigcup_{\langle e, \psi\rangle \in \overline{\sigma}} f(e)(\psi)$

(a) Type definitions

$\sqsubseteq : \overline{\Sigma} \times \overline{\Sigma} \to Bool$
$\overline{\sigma} \sqsubseteq \overline{\sigma'} := \textbf{if } (\overline{\sigma} \subseteq \overline{\sigma'}) \textbf{ then true else false}$

$\sqcup : \overline{\Sigma} \times \overline{\Sigma} \to \overline{\Sigma}$
$\overline{\sigma} \sqcup \overline{\sigma'} := \overline{\sigma} \cup \overline{\sigma'}$

$\bot : \overline{\Sigma} := \emptyset$
$\top : \overline{\Sigma} := Exp^{-} \times \overline{\Psi}$

(b) Lattice operations for $\overline{\Sigma}$

Fig. 5. λ_S monadic concrete interpreter

$\overline{m}(A) := \overline{\Psi} \to \mathcal{P}(A \times \overline{\Psi})$

$\mathsf{return}(x)(\psi) := \{\langle x, \psi\rangle\}$

$\mathsf{bind}(X)(f)(\psi) := \bigcup_{\langle x,\psi'\rangle \in X(\psi)} f(x)(\psi')$

$\mathsf{get\text{-}Env}(\langle \rho, s, ka, ks, t\rangle) := \{\langle \rho, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}Env}(\rho')(\langle \rho, s, ka, ks, t\rangle) := \{(\langle 1, \langle \rho', s, ka, ks, t\rangle\rangle\}$

$\mathsf{get\text{-}Store}(\langle \rho, s, ka, ks, t\rangle) := \{\langle s, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}Store}(s')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s', ka, ks, t\rangle\rangle\}$

$\mathsf{get\text{-}KAddr}(\langle \rho, s, ka, ks, t\rangle) := \{\langle ka, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}KAddr}(ka')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s, ka', ks, t\rangle\rangle\}$

$\mathsf{get\text{-}KStore}(\langle \rho, s, ka, ks, t\rangle) := \{\langle ks, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}KStore}(ks')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s, ka, ks', t\rangle\rangle\}$

$\mathsf{get\text{-}Time}(\langle \rho, s, ka, ks, t\rangle) := \{\langle t, \langle \rho, s, ka, ks, t\rangle\rangle\}$

$\mathsf{put\text{-}Time}(t')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho', s, ka, ks, t'\rangle\rangle\}$

$\mathsf{mzero}(\psi) := \{\}$

$(X_1 \langle + \rangle X_2)(\psi) := X_1(\psi) \sqcup X_2(\psi)$

(c) Monad definition

$$\begin{array}{lcl} \overline{[\![\cdot]\!]_m} & : & Exp^- \to \overline{\Sigma} \\ \overline{[\![e]\!]_m} & := & \mathbf{lfp}\ \lambda(x).\ x \sqcup \overline{\mathsf{init\text{-}States}}(e) \\ & & \qquad \sqcup\ (\gamma^{\overline{\Sigma} \leftrightarrow \overline{m}}(\overline{\mathsf{step}^{\mathsf{m}}}))(x) \end{array}$$

(d) Collecting semantics

Fig. 5. λ_S monadic concrete interpreter

$$
\begin{array}{l}
t \in \widehat{Time} := Exp^{*k} \\
\mathsf{tick}(\langle e, t\rangle) := \lfloor e :: t \rfloor_k \\
\\
\mathbb{Z}^\infty := \mathbb{Z} \cup \{-\infty, \infty\} \\
v \in \widehat{Val} := \mathcal{P}(\widehat{Clo}) \times ((\mathbb{Z}^\infty \times \mathbb{Z}^\infty) \cup \{\bot\}) \\
\text{int-I}(i) := \{(i, i)\} \\
\text{if0-E}(v) := \{\textbf{true} \mid v.2 \neq \bot \wedge (v.2).1 \leq 0 \leq (v.2).2\} \\
\qquad \cup \{\textbf{false} \mid v.2 = \bot \vee (v.2).1 \neq 0 \vee (v.2).2 \neq 0\} \\
\text{clo-I}(c) := \{c\} \\
\text{clo-E}(v) := \{c \mid c \in v.1\} \\
\widehat{[\![+]\!]}_{m\delta}(\langle v, v'\rangle) := \langle v.1 \cup v'.1, \langle v.2.1 + v'.2.1, v.2.2 + v'.2.2\rangle\rangle \\
\widehat{[\![-]\!]}_{m\delta}(\langle v, v'\rangle) := \langle v.1 \cup v'.1, \langle v.2.1 - v'.2.2, v.2.2 - v'.2.1\rangle\rangle
\end{array}
$$

$$
\begin{array}{l}
a \in \widehat{Addr} := Var \times \widehat{Time} \\
\rho \in \widehat{Env} := Var \rightharpoonup \widehat{Addr} \\
s \in \widehat{Store} := \widehat{Addr} \rightharpoonup \widehat{Val} \\
kf \in \widehat{KFrame} := Frame \times \widehat{Env} \\
ka \in \widehat{KAddr} := \widehat{Time} \\
ks \in \widehat{KStore} := \widehat{KAddr} \rightharpoonup \mathcal{P}(\widehat{KFrame} \times \widehat{KAddr}) \\
c \in \widehat{Clo} ::= \langle \boldsymbol{\lambda}(x).e, \rho\rangle
\end{array}
$$

$$
\begin{array}{l}
\psi \in \widehat{\Psi} := \widehat{Env} \times \widehat{Store} \times \widehat{KAddr} \times \widehat{KStore} \times \widehat{Time} \\
\sigma \in \widehat{\Sigma} := \mathcal{P}(Exp^- \times \widehat{\Psi}) \\
\widehat{\text{init-States}}(e) := \alpha(\text{init-States}(e))
\end{array}
$$

$$
\begin{array}{l}
\alpha^{\widehat{\Sigma} \hookleftarrow \widehat{m}} : (\widehat{\Sigma} \to \widehat{\Sigma}) \to (Exp^- \to \widehat{m}(Exp^-)) \\
\alpha^{\widehat{\Sigma} \hookleftarrow \widehat{m}}(f)(e)(\psi) := f(\{\langle e, \psi\rangle\}) \\
\gamma^{\widehat{\Sigma} \hookleftarrow \widehat{m}} : (Exp^- \to \widehat{m}(Exp^-)) \to (\widehat{\Sigma} \to \widehat{\Sigma}) \\
\gamma^{\widehat{\Sigma} \hookleftarrow \widehat{m}}(f)(\widehat{\sigma}) := \bigcup_{\langle e, \psi\rangle \in \widehat{\sigma}} f(e)(\psi)
\end{array}
$$

(a) Type definitions

$$
\begin{array}{l}
\sqsubseteq : \widehat{\Sigma} \times \widehat{\Sigma} \to Bool \\
\widehat{\sigma} \sqsubseteq \widehat{\sigma'} := \\
\textbf{if } (\forall \sigma \in \widehat{\sigma}. \exists \sigma' \in \widehat{\sigma'}.\ \sigma \,\tilde{\sqsubseteq}\, \sigma') \textbf{ then true else false}
\end{array}
$$

$$
\begin{array}{l}
\tilde{\sqsubseteq} : (Exp^- \times \widehat{\Psi}) \times (Exp^- \times \widehat{\Psi}) \to Bool \\
\langle e, \rho, s, ka, ks, t\rangle \,\tilde{\sqsubseteq}\, \langle e', \rho', s', ka', ks', t'\rangle := \\
\quad \textbf{if } \left(\begin{array}{c} e = e' \wedge ka = ka' \wedge t = t' \wedge \rho = \rho' \\ \wedge\ (\forall a \in s.s(a) \sqsubseteq s'(a)) \\ \wedge\ (\forall ka \in ks.ks(ka) \subseteq ks'(ka)) \end{array} \right) \\
\quad \textbf{then true else false}
\end{array}
$$

$$
\begin{array}{l}
\sqcup : \widehat{\Sigma} \times \widehat{\Sigma} \to \widehat{\Sigma} \\
\widehat{\sigma} \sqcup \widehat{\sigma'} := \widehat{\sigma} \cup \widehat{\sigma'}
\end{array}
$$

$$
\begin{array}{l}
\bot : \widehat{\Sigma} := \emptyset \\
\top : \widehat{\Sigma} := \top
\end{array}
$$

(b) Lattice operations for $\widehat{\Sigma}$

Fig. 6. λ_S monadic abstract interpreter for interval analysis

$$\widehat{m}(A) := \widehat{\Psi} \to \mathcal{P}(A \times \widehat{\Psi})$$

$$\mathsf{return}(x)(\psi) := \{\langle x, \psi\rangle\}$$

$$\mathsf{bind}(X)(f)(\psi) := \bigcup_{\langle x,\psi'\rangle \in X(\psi)} f(x)(\psi')$$

$$\text{get-Env}(\langle \rho, s, ka, ks, t\rangle) := \{\langle \rho, \langle \rho, s, ka, ks, t\rangle\rangle\}$$

$$\text{put-Env}(\rho')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho', s, ka, ks, t\rangle\rangle\}$$

$$\text{get-Store}(\langle \rho, s, ka, ks, t\rangle) := \{\langle s, \langle \rho, s, ka, ks, t\rangle\rangle\}$$

$$\text{put-Store}(s')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s', ka, ks, t\rangle\rangle\}$$

$$\text{get-KAddr}(\langle \rho, s, ka, ks, t\rangle) := \{\langle ka, \langle \rho, s, ka, ks, t\rangle\rangle\}$$

$$\text{put-KAddr}(ka')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s, ka', ks, t\rangle\rangle\}$$

$$\text{get-KStore}(\langle \rho, s, ka, ks, t\rangle) := \{\langle ks, \langle \rho, s, ka, ks, t\rangle\rangle\}$$

$$\text{put-KStore}(ks')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho, s, ka, ks', t\rangle\rangle\}$$

$$\text{get-Time}(\langle \rho, s, ka, ks, t\rangle) := \{\langle t, \langle \rho, s, ka, ks, t\rangle\rangle\}$$

$$\text{put-Time}(t')(\langle \rho, s, ka, ks, t\rangle) := \{\langle 1, \langle \rho', s, ka, ks, t'\rangle\rangle\}$$

$$\mathsf{mzero}(\psi) := \{\}$$

$$(X_1 \langle + \rangle X_2)(\psi) := X_1(\psi) \sqcup X_2(\psi)$$

(c) Monad definition

$$\alpha : \overline{\Sigma} \to \widehat{\Sigma}$$
$$\alpha(\overline{\sigma}) := \{\alpha(\sigma) \mid \sigma \in \overline{\sigma}\}$$

$$\alpha : Exp^- \times \overline{\Psi} \to Exp^- \times \widehat{\Psi}$$
$$\alpha(\langle e, \rho, s, ka, ks, t\rangle) := \langle e, \alpha(\rho), \alpha(s), \alpha(ka), \alpha(ks), \alpha(t)\rangle$$

$$\alpha : \overline{Env} \to \widehat{Env}$$
$$\alpha(\rho) := \{\langle x, \langle x, \lfloor \rho(x).2 \rfloor_k\rangle\rangle \mid x \in \rho\}$$

$$\alpha : \overline{Store} \to \widehat{Store}$$
$$\alpha(s) := \lambda(\widehat{a}).\ \bigsqcup_{\alpha(a)=\widehat{a} \wedge a \in s} \alpha(s(a))$$

$$\alpha : \overline{KAddr} \to \widehat{KAddr}$$
$$\alpha(ka) := \lfloor ka \rfloor_k$$

$$\alpha : \overline{KStore} \to \widehat{KStore}$$
$$\alpha(ks) := \lambda(\widehat{ka}).\ \bigcup_{\alpha(ka)=\widehat{ka} \wedge ka \in ks} \alpha(ks(ka))$$

$$\alpha : \overline{Time} \to \widehat{Time}$$
$$\alpha(t) := \lfloor t \rfloor_k$$

(d) Abstraction map α from $\overline{\Sigma}$ to $\widehat{\Sigma}$

$$\begin{array}{lcl} \widehat{[\![\cdot]\!]}_m & : & Exp^- \to \widehat{\Sigma} \\ \widehat{[\![e]\!]}_m & := & \mathbf{lfp}\ \lambda(x).\ x \sqcup \widehat{\mathsf{init\text{-}States}}(e) \\ & & \qquad \sqcup\ (\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}}(\widehat{\mathsf{step}^{\mathsf{m}}}))(x) \end{array}$$

(e) Abstract semantics

Fig. 6. λ_S monadic abstract interpreter for interval analysis

Figure 6a includes all the typeclass definitions, except the monad definition. For Van Horn-Might abstract machines, the notion of time plays a key role in dictating the abstract machine behavior. In particular, the set of addresses available in the value and continuation stores depends on the definition of time. For the abstract interpreter, we want to finitize the set of available addresses, and this is achieved by restricting $\widehat{Time}$ to sequences of upto k expressions (with the set of syntactic expressions contained in a program being finite), as opposed to sequences of unbounded length for concrete interpreters. Values ($\widehat{Val}$) are defined as a pair of a set of closures and an integer interval. Note that we extend the set of integers $\mathbb{Z}$ to $\mathbb{Z}^{\infty}$ that includes $\{-infty, infty\}$. The top element of the set of intervals is defined as $\langle -\infty, \infty \rangle$ while bottom is defined by a special element $\bot$. The reason for defining values as pairs of closures and intervals is that, due to the finite number of addresses available in the store, it is possible for a particular location to be mapped to values of both these types. The operations defined for $\widehat{Val}$ are self-explanatory though we make a quick comment on notation - the projection of the i^{th} element of a tuple t is written as $t.i$, with indices starting from 1. All the other definitions in Figure 6a are straightforward. Note that the abstract version of init-States ($\widehat{\text{init-States}}$) applies the abstraction map α, defined in Figure 6d to set of initial states constructed by $\overline{\text{init-States}}$. In the abstract setting, this set of initial states only contains a single element, irrespective of whether the expression is closed, or if it has free variables of type $\mathbb{Z}$.

Figure 6b defines the lattice operations for $\widehat{\Sigma}$. An element $\widehat{\sigma}$ of $\widehat{\Sigma}$ is itself a set of abstract states. The lattice order operation ($\sqsubseteq$) is defined such that $\widehat{\sigma}$ is "less than" $\widehat{\sigma'}$ if for every element $\sigma \in \widehat{\sigma}$, there exists at least one element $\sigma' \in \widehat{\sigma'}$ such that $\sigma \tilde{\sqsubseteq} \sigma'$. $\tilde{\sqsubseteq}$ is itself defined such that σ is "less than" σ' if and only if σ and σ' the expression, the environment, the address of the next continuation, and the time components are the same, and for every address in σ's store s that is mapped to a value v, the same address in store s' in σ' is mapped to a value v' that is at least as large as v, and similarly, for every address in σ's continuation store ks that is mapped to a set X of continuations, the same address in store ks' in σ' is mapped to a set X' that is equal to X or a superset of X. A comment on the notation - we do not use distinct symbols to represent the lattice operations for different lattices, but the lattice being considered should be clear from the context. The bottom of $\widehat{\Sigma}$ lattice is just the empty set while the top is defined by the special element $\top$.

Figure 6c defines the monad for the abstract interpreter and figure 6d defines the abstraction map from $\overline{\Sigma}$ to $\widehat{\Sigma}$. The abstract version of an element $\overline{\sigma} \in \overline{\Sigma}$ is obtained by abstracting each element $\sigma \in \overline{\sigma}$. In the same way that we do not notationally distinguish between lattice operations for different lattices, we do not notationally distinguish between the different abstraction operations, but the types involved should be clear from the context. As one would expect, abstracting a concrete abstract machine state involves abstracting every element of the state tuple. Environment abstraction requires abstracting the addresses that variables are mapped to. These addresses are pairs of variable names and times, and an abstract version of time t requires truncating the sequence of expressions to the latest k expressions. A value store is abstracted by first abstracting the addresses in the store, and then joining all the values that map to the same address. Similarly, continuation stores are abstracted by abstracting the addresses, and then taking a union of all the sets of continuations that map to the same address. Finally, the abstract semantics of a program in λ_S are defined as the least fixed point of a function that uses the abstract versions of the init-States and the step$^{\text{m}}$ functions, i.e., $\widehat{\text{init-States}}$ and $\widehat{\text{step}^{\text{m}}}$.

We next state two propositions relating the λ_S monadic abstract interpreter to the monadic concrete interpreter. As is typical in the theory of abstract interpretation, we would like to state that the abstract interpreter is sound with respect to the concrete interpreter.

Proposition 2. *(Soundness of* $\widehat{\mathsf{step}^{\mathsf{m}}}$ *with respect to* $\overline{\mathsf{step}^{\mathsf{m}}}$*)*
$\forall \overline{\sigma} \in \overline{\Sigma}.\ \alpha((\gamma^{\overline{\Sigma} \leftrightarrow \overline{m}}(\overline{\mathsf{step}^{\mathsf{m}}}))(\overline{\sigma})) \sqsubseteq (\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}}(\widehat{\mathsf{step}^{\mathsf{m}}}))(\alpha(\overline{\sigma}))$

Proposition 2 relates the concrete $\overline{\mathsf{step}^{\mathsf{m}}}$ function to the abstract $\widehat{\mathsf{step}^{\mathsf{m}}}$ function. In particular, for every element $\overline{\sigma} \in \overline{\Sigma}$, we want the abstraction of the result of applying $\overline{\mathsf{step}^{\mathsf{m}}}$ to $\overline{\sigma}$ to be "less than" the result of applying $\widehat{\mathsf{step}^{\mathsf{m}}}$ to $\alpha(\overline{\sigma})$. This proposition does not directly relate the concrete and abstract semantics of λ_S (which involve computing least fixed points), but it can help us prove the soundness of $\widehat{[\![\cdot]\!]}_m$ with respect to $\overline{[\![\cdot]\!]}_m$. We can prove this result by performing a case analysis on the structure of λ_S expressions, where the cases are the same as that considered by $\mathsf{step}^{\mathsf{m}}$. We do not present the proof here.

Proposition 3 states the soundness relationship between the abstract and the concrete semantics. In particular, for every λ_S program $e \in Exp^{-}$, it states that the result of abstracting the meaning of the program, as defined by the concrete semantics $\overline{[\![e]\!]}_m$, is less than the meaning defined by the abstract semantics $\widehat{[\![e]\!]}_m$. In other words, the program semantic invariant computed using $\widehat{[\![\cdot]\!]}_m$ can be safely used in proofs of program correctness.

Proposition 3. *(Soundness of* $\widehat{[\![\cdot]\!]}_m$ *with respect to* $\overline{[\![\cdot]\!]}_m$*)*
$\forall e \in Exp^{-}.\ \alpha(\overline{[\![e]\!]}_m) \sqsubseteq \widehat{[\![e]\!]}_m$

Proof. We only present an informal proof sketch. From a proof of proposition 2, a proof of this proposition can be constructed in standard manner using the fixed point transfer theorem from [Cousot and Cousot 1979]. In particular, the monotonicity of the functions $\lambda(x).\ x \sqcup \widehat{\mathsf{init\text{-}States}}(e) \sqcup (\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}}(\widehat{\mathsf{step}^{\mathsf{m}}}))(x)$ and $\lambda(x).\ x \sqcup \overline{\mathsf{init\text{-}States}}(e) \sqcup (\gamma^{\overline{\Sigma} \leftrightarrow \overline{m}}(\overline{\mathsf{step}^{\mathsf{m}}}))(x)$, combined with proposition 2 and the Knaster–Tarski theorem fixed point theorem yields the required result. ■

4 OBSERVATIONAL ABSTRACT INTERPRETERS

In the previous sections, we have defined the language $\lambda_S(\lambda_{SA})$, and have discussed the construction of a monadically-structured interpreter for λ_S. This monadic interpreter is parameterized, i.e., the types of data accessed by the interpreter are defined using typeclass-like constructions. By suitably instantiating these typeclasses, one can recover the concrete semantics of the language. Additionally, one can also instantiate these typeclasses to yield an abstract interpreter (in our case, an abstract interpreter capable of interval analysis). From the perspective of proofs about program judgments, the monadic interpreter is a meta-theoretic construction for computing semantic program invariants. The abstract semantics or abstract meaning of a program, computed with a monadic abstract interpreter, is a semantic invariant of the program and a simplified representation (informally, containing lesser information) of the concrete program meaning.

We want to combine semantic invariant based reasoning, with reasoning hypothetically about program via run time/dynamic checks. Moreover, we want to use observations about program behavior for inferring the hypotheses. We achieve this by extending the monadic interpreter design, proposing a new meta-theoretic construction for reasoning about programs, that we refer to as *observational abstract interpreters*. There are two main reasons motivating our construction of observational abstract interpreters: (i) past work has investigated various combinations of invariant-based reasoning, hypothetical reasoning, and observational reasoning about programs, but there has been an absence of formal investigation of approaches combining these three reasoning styles. A precise formulation of a combined approach can bring greater clarity about the design space of algorithms for finding proofs of program judgments. (ii) hypothetical reasoning about programs using observations about their behavior can help us focus the program proof effort towards the

$\mathring{Obs}$

$\underline{\mathring{m}_o}$
return : $\forall A.A \to \mathring{m}_o(A)$
bind : $\forall A, B.\mathring{m}_o(A) \to (A \to \mathring{m}_o(B)) \to \mathring{m}_o(B)$
get-Env : $\mathring{m}_o(Env)$
put-Env : $Env \to \mathring{m}_o(1)$
get-Store : $\mathring{m}_o(\mathring{Store})$
put-Store : $\mathring{Store} \to \mathring{m}_o(1)$
get-KAddr : $\mathring{m}_o(\mathring{KAddr})$
put-KAddr : $\mathring{KAddr} \to \mathring{m}_o(1)$
get-KStore : $\mathring{m}_o(\mathring{KStore})$
put-KStore : $\mathring{KStore} \to \mathring{m}_o(1)$
get-Time : $\mathring{m}_o(\mathring{Time})$
put-Time : $\mathring{Time} \to \mathring{m}_o(1)$
mzero : $\forall A.\mathring{m}_o(A)$
$\cdot\langle+\rangle\cdot$: $\forall A.\mathring{m}_o(A) \times \mathring{m}_o(A) \to \mathring{m}_o(A)$
obs-Store : $Exp^- \times Var \times \mathring{Val} \times \mathring{Obs} \to \mathring{m}_o(\mathring{Val})$

$\alpha^{\mathring{\Sigma} \leftrightarrow \mathring{m}_o}$: $(\mathring{Obs} \to (\mathring{\Sigma} \to \mathring{\Sigma})) \to (\mathring{Obs} \to (Exp^- \to \mathring{m}(Exp^-)))$
$\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}_o}$: $(\mathring{Obs} \to (Exp^- \to \mathring{m}(Exp^-))) \to (\mathring{Obs} \to (\mathring{\Sigma} \to \mathring{\Sigma}))$

(a) Type definitions

$\text{step}_o^{\text{m}} : Exp^- \times Obs \to \mathring{m}_o(Exp^-)$
$\text{step}_o^{\text{m}}(e)$:= **do**
  $\rho \leftarrow$ get-Env
  $e'' \leftarrow$ **case** e **of**
    $e_1 \odot e_2 \to \text{tick}^{\text{m}}(e)$; push($\langle \square \odot e_2, \rho \rangle$); return($e_1$)
    if0(e_1){e_2}{e_3} $\to$ **do**
      $\text{tick}^{\text{m}}(e)$; push($\langle$**if0**($\square$){$e_2$}{$e_3$}, $\rho\rangle$); return(e_1)
    $a \to$ **do**
      $v \leftarrow [\![a]\!]_{mA}$; $fr \leftarrow$ pop
      case fr **of**
        $\langle \square \odot e', \rho' \rangle \to$ **do**
          $\text{tick}^{\text{m}}(e)$; put-Env($\rho'$); push($\langle v \odot \square, \rho \rangle$); return($e'$)
        $\langle v'@\square, \rho' \rangle \to$ **do**
          $\text{tick}^{\text{m}}(e)$; $t \leftarrow$ get-Time; $s \leftarrow$ get-Store
          $\langle \boldsymbol{\lambda}(x).e', \rho'' \rangle \leftarrow \uparrow_p(\text{clo-E}(v'))$
          put-Env($\rho''[x \mapsto \langle x, t \rangle]$)
          $v' \leftarrow$ obs-Store($\langle e', x, v, o \rangle$)
          put-Store($s \sqcup [\langle x, t \rangle \mapsto v']$); return($e'$)
        $\langle v' \oplus \square, \rho' \rangle \to \text{tick}^{\text{m}}(e)$; return($[\![\oplus]\!]_{m\delta}\langle\langle v', v \rangle\rangle$)
        $\langle$**if0**($\square$){e_1}{e_2}, $\rho'\rangle \to$ **do**
          $\text{tick}^{\text{m}}(e)$; put-Env($\rho'$); $b \leftarrow \uparrow_p(\text{if0-E}(v))$; refine($\langle a, b \rangle$)
          if (b) **then** return(e_1) **else** return(e_2)
        $\bot \to$ return(e)
  return(e'')

(b) Step function

$$[\![\cdot]\!]_{mO} \quad : \quad Exp^- \times Obs \to \mathring{\Sigma}_o$$
$$[\![e]\!]_{mO}(o) \quad := \quad \textbf{lfp}\ \lambda(x).\ x \sqcup \text{init-States}(e) \sqcup (\gamma^{\mathring{\Sigma} \leftrightarrow \mathring{m}_o}(\text{step}_O^{\text{m}}))(x)(o)$$

(c) Collecting semantics

Fig. 7. λ_S observational interpreter

observed or common program behaviors, potentially making the the search for program proofs cheaper, at the cost of dynamic/run time checks.

In Figure 7, we present an observational abstract interpreter for λ_S. This interpreter is monadically structured, and designed such that while the semantic invariant, i.e., the program semantics, is being computed, the interpreter can read data representing observations about program behavior, use these observations to make hypotheses about program behavior, and accordingly update the state of the interpreter. Moreover, the validity of these hypotheses statically, and instead, we embed dynamic checks in to the λ_S programs, producing λ_{SA} programs. In the process of designing an observational abstract interpreters, following are the main questions that we were forced to address,

- What is the form of the observational data about programs? What aspects of program behavior does it capture?
- How do we infer the hypotheses using the observational data? Moreover, how do we avoid inferring too many hypotheses/dynamic checks, and how do we ensure that the inferred hypotheses are not overly restrictive, such that the program fails to satisfy the dynamic checks in most cases?
- How do we translate the hypotheses in to dynamic checks embedded in the program?

For the first question, the observational abstract interpreter design in Figure 7 assumes that the observational data is drawn from the collecting semantics, i.e., the set of reachable abstract machine states, of a λ_S program. However, the exact form of the observations is left unspecified (we give a specific definition for the observational interval analysis defined in Figure 8). Using observations about program inputs in order to infer program pre-conditions is not uncommon [Padhi et al. 2016], but our design allows observations at any program point, and about any component of the abstract machine. In Figure 7, the typeclass $\mathring{Obs}$, with no constraints, represent the types of observations. We use the blue background to highlight parts of the observational interpreter design that are unique. We do not show any type definitions besides the monad typeclasses and elide the helper functions because these are similarly to the definitions in Figure 4.

To address the second question, we extend the monad typeclass with the operation obs-Store as shown in Figure 7a. This operation requires that a 4-tuple comprising of the current expression being evaluated, a variable name, the value associated with the variable, and the observational data is passed as an argument. We also modify step^{m} such that whenever the term in the argument position of a function application is evaluated to a value and the next step of the evaluation is to actually apply the function to this value, the observational interpreter first invokes the obs-Store operation with the name of the argument (say x) and it's evaluated value (say v). Next, instead of substituting x with v in the function, we substitute it with the value (say v') returned by obs-Store (say v'). The hypothesis that the value of x is v' instead of v is the only form of hypothesis that the observational interpreter is allowed to make. The mechanism for computing v' is hidden behind the monadic interface. Notice that the observational data is passed as an argument to the $\text{step}^{\text{m}}_{O}$ function. This is also forces us to change the type definitions of $\alpha^{\mathring{\Sigma}\leftrightarrow\mathring{m}_o}$ and $\gamma^{\mathring{\Sigma}\leftrightarrow\mathring{m}_o}$ (Figure 7a) that map between the monadic step function and transition function of type $\mathring{\Sigma} \to \mathring{\Sigma}$. Moreover, the observational collecting semantics ($[\![\cdot]\!]_{mO}$) are also modified to accept observations as an argument.

We address the third question in the specific context of an observational abstract interpreter for an interval analysis of λ_S programs in the next section.

4.1 Observational Interval Analysis for λ_S

We instantiate the generic observational interpreter for λ_S so as to yield an observational abstract interpreter for interval analysis of λ_S programs as described in Figure 8. The type definitions are presented in Figure 8a. Notice that observations (*Obs*) are defined as a partial map from labels to

$o \in Obs := Lbl \rightharpoonup (Var \rightharpoonup \mathcal{P}(\mathbb{Z}))$
$h \in Hyp := Lbl \rightharpoonup (Var \rightharpoonup \widehat{Val})$
$\psi \in \widehat{\Psi} := \widehat{Env} \times \widehat{Store} \times \widehat{KAddr} \times \widehat{KStore} \times \widehat{Time}$
$\widehat{\sigma}_o \in \widehat{\Sigma}_o := \mathcal{P}(Exp^- \times \widehat{\Psi}) \times Hyp$
$\widehat{\text{init-States}}_o(e) := \langle \alpha(\widehat{\text{init-States}}(e)), \bot \rangle$

$\alpha^{\widehat{\Sigma} \leftrightarrow \widehat{m}_o} : (Obs \to (\widehat{\Sigma}_o \to \widehat{\Sigma}_o)) \to (Obs \to (Exp^- \to \widehat{m}_o(Exp^-)))$
$\alpha^{\widehat{\Sigma} \leftrightarrow \widehat{m}_o}(f)(o)(e)(\langle \psi, h \rangle) := f(o)(\langle \{\langle e, \psi \rangle\}, h \rangle)$
$\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}_o} : (Obs \to (Exp^- \to \widehat{m}_o(Exp^-))) \to (Obs \to (\widehat{\Sigma}_o \to \widehat{\Sigma}_o))$
$\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}_o}(f)(o)(\widehat{\sigma}_o) :=$ **let** $\langle X, h \rangle := \widehat{\sigma}_o$ **in**
$\quad \langle \bigcup_{\langle e, \psi \rangle \in X} f(o)(e)(\langle \psi, h \rangle).1, \bigsqcup_{\langle e, \psi \rangle \in X} f(o)(e)(\langle \psi, h \rangle).2 \rangle$

(a) Type definitions

$\sqsubseteq : \widehat{\Sigma}_o \times \widehat{\Sigma}_o \to Bool$
$\widehat{\sigma}_o \sqsubseteq \widehat{\sigma'}_o :=$
if $((\forall \sigma \in \widehat{\sigma}_o.1. \exists \sigma' \in \widehat{\sigma'}_o.1.\ \sigma \,\tilde{\sqsubseteq}_o\, \sigma') \wedge (\widehat{\sigma}_o.2 \sqsubseteq \widehat{\sigma'}_o.2))$
then true else false

$\tilde{\sqsubseteq}_o : (Exp^- \times \widehat{\Psi}) \times (Exp^- \times \widehat{\Psi}) \to Bool$
$\tilde{\sqsubseteq}_o := \tilde{\sqsubseteq}$

$\sqsubseteq : Hyp \times Hyp \to Bool$
$h \sqsubseteq h' :=$ **if** $(\forall l \in h. \forall x \in h(l).h(l)(x) \sqsupseteq h'(l)(x))$
then true else false

$\sqcup : \widehat{\Sigma}_o \times \widehat{\Sigma}_o \to \widehat{\Sigma}_o$
$\widehat{\sigma}_o \sqcup \widehat{\sigma'}_o := \langle \widehat{\sigma}_o.1 \cup \widehat{\sigma'}_o.1, \widehat{\sigma}_o.2 \sqcup \widehat{\sigma'}_o.2 \rangle$

$\sqcup : Hyp \times Hyp \to Hyp$
$h \sqcup h' :=$
let $f := (\lambda(x).$ **if** $(x \in h(l) \wedge x \in h'(l))$
then if $(h(l)(x) = h'(l)(x))$ **then** $h(l)(x)$ **else** $\bot$
else if $(x \in h(l))$ **then** $h(l)(x)$ **else** $h'(l)(x))$ **in**
let $g = (\lambda(x).$ **if** $(l \in h \wedge x \in h(l))$
then $h(l)(x)$ **else if** $(l \in h' \wedge x \in h'(l))$ **then** $h'(l)(x))$ **in**
$\{f \mid l \in h \wedge l \in h'\} \cup \{g \mid l \in h \text{ xor } l \in h'\}$

$\bot : \widehat{\Sigma}_o := \langle \emptyset, \bot \rangle$
$\top : \widehat{\Sigma}_o := \langle \top, \top \rangle$

(b) Lattice operations for $\widehat{\Sigma}_o$

Fig. 8. λ_S observational abstract interpreter for interval analysis

$$\widehat{m}_o(A) := \widehat{\Psi} \times Hyp \to \mathcal{P}(A \times \widehat{\Psi}) \times Hyp$$

obs-Store : $Exp^- \times Var \times \widehat{Val} \to \widehat{m}_o(\widehat{Val})$
obs-Store$(\langle e, x, v, o\rangle)(\langle \psi, h\rangle) :=$
if (get-Label$(e) \in h \wedge x \in h($get-Label$(e))$) **then** {
  $\langle\{\langle\langle v.1, h(\text{get-Label}(e))(x)\rangle, \psi\rangle\}, h\rangle$
} **else if** ((get-Label$(e) \in o) \wedge (x \in o($get-Label$(e)))$) **then** {
  let $v_O := \alpha(o(\text{get-Label}(e))(x))$ **in**
  let distance $:= d(v.2, v_O)$ **in**
  if (distance $\geq \omega \wedge v_O \sqsubseteq v.2$) **then** {
    $\langle\{\langle\langle v.1, v_O\rangle, \psi\rangle\}, h \sqcup [\text{get-Label}(e) \mapsto [x \mapsto \langle\emptyset, v_O\rangle]]\rangle$
  } **else** $\langle\{\langle v, \psi\rangle\}, h\rangle$
} **else** $\langle\{\langle v, \psi\rangle\}, h\rangle$

return$(x)(\langle\psi, h\rangle) := \langle\{\langle x, \psi\rangle\}, h\rangle$

bind$(X)(f)(\langle\psi, h\rangle) :=$ **let** $\langle Y, h'\rangle := X(\langle\psi, h\rangle)$ **in**
$\bigcup_{\langle x, \psi'\rangle \in Y} f(x)(\langle\psi', h'\rangle)$

get-Env$(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle\rho, \langle\rho, s, ka, ks, t\rangle\rangle\}, h\rangle$

put-Env$(\rho')(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle 1, \langle\rho', s, ka, ks, t\rangle\rangle\}, h\rangle$

get-Store$(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle s, \langle\rho, s, ka, ks, t\rangle\rangle\}, h\rangle$

put-Store$(s')(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle 1, \langle\rho, s', ka, ks, t\rangle\rangle\}, h\rangle$

get-KAddr$(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle ka, \langle\rho, s, ka, ks, t\rangle\rangle\}, h\rangle$

put-KAddr$(ka')(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle 1, \langle\rho, s, ka', ks, t\rangle\rangle\}, h\rangle$

get-KStore$(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle ks, \langle\rho, s, ka, ks, t\rangle\rangle\}, h\rangle$

put-KStore$(ks')(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle 1, \langle\rho, s, ka, ks', t\rangle\rangle\}, h\rangle$

get-Time$(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle t, \langle\rho, s, ka, ks, t\rangle\rangle\}, h\rangle$

put-Time$(t')(\langle\langle\rho, s, ka, ks, t\rangle, h\rangle) := \langle\{\langle 1, \langle\rho', s, ka, ks, t'\rangle\rangle\}, h\rangle$

mzero$(\langle\psi, h\rangle) := \langle\{\}, h\rangle$

$(X_1\langle+\rangle X_2)(\langle\psi, h\rangle) := X_1(\langle\psi, h\rangle) \cup X_2(\langle\psi, h\rangle)$

(c) Monad definition

$$\begin{array}{rcl} \alpha & : & \overline{\Sigma} \to \widehat{\Sigma}_o \\ \alpha(\overline{\sigma}) & := & \langle\{\alpha(\sigma) \mid \sigma \in \overline{\sigma}\}, \bot\rangle \end{array}$$

(d) Abstraction map α from $\overline{\Sigma}$ to $\widehat{\Sigma}_o$

$$\begin{array}{rcl} \widehat{[\![\cdot]\!]}_{mO} & : & Exp^- \times Obs \to \widehat{\Sigma}_o \\ \widehat{[\![e]\!]}_{mO}(o) & := & \textbf{lfp}\ \lambda(x).\ x \sqcup \widehat{\text{init-States}}_o(e) \\ & & \sqcup\ (\gamma^{\widehat{\Sigma} \leftrightarrow \widehat{m}}(\widehat{\text{step}^{\overline{m}}_O}))(x)(o) \end{array}$$

(e) Abstract semantics

Fig. 8. λ_S observational abstract interpreter for interval analysis

$$
\begin{array}{rcl}
i \in \mathit{Intrvl} & : & \mathbb{Z}^{\infty} \times \mathbb{Z}^{\infty} \\
d & : & \mathit{Intrvl} \times \mathit{Intrvl} \to \mathbb{R} \cup \{-\infty, \infty\} \\
d(\langle i, i' \rangle) & := & \textbf{let } X := \{-\infty, \infty\} \textbf{ in} \\
 & & \textbf{if } (i.1 \in X \vee i.2 \in X \vee i = \bot \vee i'.1 \in X \vee i'.2 \in X \vee i' = \bot) \\
 & & \textbf{then } \infty \\
 & & \textbf{else } \max(\langle |i.1 - i'.1|, |i.2 - i'.2| \rangle)
\end{array}
$$

Fig. 9. Metric structure on intervals

$$
\begin{array}{lcl}
\mathsf{embed}_{\mathsf{t}} & : & \mathit{Exp}^{-}{}_{\lambda_S} \times (\mathit{Lbl} \times \mathit{Var} \times \widehat{\mathit{Val}}) \to \mathit{Exp}^{-}{}_{\lambda_{SA}} \\
\mathsf{embed}_{\mathsf{t}}(\langle e, \langle l, x, v \rangle \rangle) & := & \textbf{let } v' := v.2 \textbf{ in} \\
 & & \textbf{let } e' := \textbf{if0}(\alpha(x) \sqsubseteq v')\{e\}\{\textbf{abort}\} \textbf{ in} \\
 & & \textbf{if } (\text{get-Label}(e) = l) \textbf{ then } e' \textbf{ else } e
\end{array}
$$

Fig. 10. Translation of λ_S programs in to λ_{SA} programs with embedded dynamic checks (assuming that $\sqsubseteq$ returns 1 for **true** and 0 for **false**)

partial maps from variables names to sets of values. We assume that program observations are recorded at the granularity of syntactic program expressions, explicitly identified by their labels. Moreover, for each expression we can record a set of observed values for any variable in scope. We also assume that only the values of type $\mathbb{Z}$ are recorded. Extending this approach to with observations of higher-order values is an interesting direction for future work. The observational abstract interpreter computes an element $\widehat{\sigma}_o \in \widehat{\Sigma}_o$, where each element $\widehat{\sigma}_o$ is a pair of a set of abstract machine states and the hypotheses map. A hypotheses map h is a partial map from labels to partial maps from variables names to abstract values. Intuitively, the interpreter can make hypotheses at the granularity of syntactic program expressions. At each program expressions, hypotheses can made about the abstract values of the variables in scope. Though the type of hypotheses maps ($\mathit{Lbl} \rightharpoonup \mathit{Var} \rightharpoonup \widehat{\mathit{Val}}$) allows assumptions about higher-order values, the observational abstract interpreter defined here only makes assumptions on $\mathbb{Z}$ values. Initially, the hypotheses map is assumed to be $\bot$, as the definition of $\widehat{\text{init-States}}_o$ shows.

Figure 8b defines the lattice operations for the lattice $\widehat{\Sigma}_o$. We draw notice to the definitions of the lattice operations for the hypotheses map. A hypotheses map h is "less than" a hypotheses map h' if for every label and variable for which h includes a hypothesis, h' has a stricter hypotheses, i.e., assumes a narrower interval of $\mathbb{Z}$. The join operation for hypotheses maps h and h' looks messy but the intuition is simple - whenever a hypothesis is defined for one map but not the other, we defer to the map with the definition, but in case both the maps have hypotheses defined for a particular combination of label and variable, then we require the two hypotheses be equal, or the join produces the bottom element of the $\widehat{\mathit{Val}}$ lattice as the joined hypothesis. The bottom element of the Hyp lattice makes no hypotheses whereas the top element of the Hyp lattice makes the strictest possible possible hypothesis for every label and variable.

Figure 8c defines the monad $\widehat{m}_o$ for observational abstract interpreters. The only interesting definition is that of obs-Store. The other monad operations are similar to the definition of the monad operations for the monadic abstract interpreter in Figure 6e. obs-Store expects a 4-tuple of expression, variable name, value, and the observations ($\langle e, x, v, o \rangle$). It extracts the label of the expression e using the get-Label function, and checks if a hypothesis has been already made for

variable x at label t, and if so, it replaces the second element of v (recall that an abstract value is a pair of a set of closures and an interval) with the hypothesis. In case there is no preexisting hypothesis, and if the observations map includes a set of observed values of x at label t, then the set of observed $\mathbb{Z}$ values is first abstracted to an interval $\widehat{v}_o$ (assumed here to bee tightest possible interval abstraction of the set of observed values, though other choices are possible). Next, the distance between the intervals $\widehat{v}$ and $v.2$ is computed. Such a distance computation is possible because we give a metric structure to the set of intervals (defined in Figure 9). Finally, if the distance is greater than a fixed constant ω (we expect value of ω to be empirically derived), and if $\widehat{v}_o \sqsubseteq v.2$, then we replace $v.2$ with $\widehat{v}_o$, and update the hypotheses map accordingly.

Figure 8d defines the abstraction map α from $\overline{\Sigma}$ to $\widehat{\Sigma}_o$. The abstraction map reuses the definition of the abstraction map from Figure 6d for the set of abstract machines states, but the hypotheses map is always assumed to be $\bot$. Finally, the observational abstract semantics, defined in Figure 8e take the standard least fixed point form, except that the observations map is expected as an input.

The metric structure on the set of intervals in defined in Figure 9. A set X has a metric structure for all elements x, y, z in X, if a function $d(\cdot,\cdot)$ producing a value of type $\mathbb{R}$ is defined for X, such that the following conditions hold true,

- $d(x,y) = 0 \iff x = y$
- $d(x,y) = d(y,x)$
- $d(x,z) \leq d(x,y) + d(y,z)$

Finally, Figure 10 describes the manner in which a hypothesis can be embedded in a λ_S program. For ease of presentation, we define a function embed_t that given a program from the set ${Exp^-}_{\lambda_S}$ of λ_S programs with free variables of type $\mathbb{Z}$, and a triple of a label, variable name, and an abstract value ($\langle l, x, v\rangle$), produces a λ_{SA} program where the original expression e is wrapped in a dynamic check. We assume here that the abstraction map α and the lattice operation $\sqsubseteq$ are computable.

Proposition 4 is a formal statement of the notion of soundness for our observational abstract interpreter, relating the observational abstract semantics ($\widehat{[\![\cdot]\!]}_{mO}$) of a λ_S program in Exp^- with the monadic concrete semantics ($\overline{[\![\cdot]\!]}_m$) of the same. Intuitively, the proposition states that for any expression $e \in Exp^-$ and for any observations map o, if we compute the observational abstract semantics of e, producing the pair $\langle\widehat{\sigma}, h\rangle$, then the λ_{SA} program e' obtained by embedding the hypotheses map h in e is such that an abstraction of the computed collecting semantics of e' is less than or equal to $\widehat{\sigma}$ extended with an **abort** abstract machine state. Note that $\overline{[\![\cdot]\!]}_m$ here denotes the monadic collecting semantics of λ_{SA} which is exactly the same as for λ_S (in Figure 5) except for the **abort** operation

Proposition 4. *(Soundness of $\widehat{[\![\cdot]\!]}_{mO}$ with respect to $\overline{[\![\cdot]\!]}_m$)*
$\forall e \in Exp^-, o \in Obs$. *If* $\langle\widehat{\sigma}, h\rangle := \widehat{[\![e]\!]}_{mO}(o)$, *then*, $\alpha(\overline{[\![\text{embed}(e,h)]\!]}_m) \sqsubseteq (\widehat{\sigma} \cup \{\langle \textbf{abort}, \top, \top, \top, \top, \top\rangle\})$

We do not present a proof of proposition 4 in this paper, though we believe that the observational abstract interpreter in Figure 8 is sound with respect to the monadic concrete semantics of λ_S. The proof is challenging primarily because the function $\gamma^{\widehat{\Sigma}\leftrightarrow\widehat{m}_o}(\text{step}^{\text{m}}_O)$ of type $\widehat{\Sigma}_o \to \widehat{\Sigma}_o$ is not monotonic.

5 RELATED WORK

There are many threads of work related to the ideas presented in this article, and we described some of these connections in Section 1. In this section, we further elaborate on three main threads of related work, namely, on the use of observations about a program for constructing program proofs, on the use of "big code", i.e. repositories of existing programs and associated metadata, for

learning statistical models that can help reason about programs, and finally, on embedding dynamic checks to aid static reasoning about program.

Using program observations for program proofs. There is a long history of using observations to make hypotheses about program behavior, and computing semantic invariants under these hypotheses [Bodden et al. 2011; Csallner et al. 2008; Devecsery et al. 2018; Dufour et al. 2007; Grech et al. 2017; Gupta et al. 1997; Kinder and Kravchenko 2012; Mock et al. 2002; Wei and Ryder 2013]. Our work on observational abstract interpreters formalizes this style of reasoning. A different line of work uses program observations to guide CEGAR algorithms in their search for an appropriate abstract semantics [Beckman et al. 2010; Gupta et al. 2009; Naik et al. 2012]. More recently, with the advances in statistical learning algorithms, a number of techniques have been proposed that eschew the use of abstract interpreters and instead use the observational data to iteratively infer (or learn) candidate invariants that, if confirmed to be invariants (typically using an SMT-like decision procedure), are used to help in the construction of program proofs [Ernst et al. 2007; Fedyukovich et al. 2017; Garg et al. 2014, 2016; Le et al. 2019; Miltner et al. 2020; Sharma and Aiken 2016; Sharma et al. 2013b,a; Si et al. 2018; Zhu et al. 2018, 2015, 2016]. Program observations have also been used to compute candidate specifications [Ammons et al. 2002; Bastani et al. 2018; Padhi et al. 2016; Sankaranarayanan et al. 2008] or types of program modules [An et al. 2011; Furr et al. 2009].

"Big code" and program reasoning. Using "big code", i.e., a dataset of programs and corresponding program metadata (like test cases, bug reports, program analysis results, etc.), one can construct statistical models about the nature of programs that humans write, and use these models to help reason about programs. With the rapid advances in computational statistical modeling and machine learning in recent years, this style of reasoning has become increasingly feasible. We give a small sampling here of the literature on using statistical models for reasoning about programs. Statistical models have been used to, (i) help in the computation of program invariants by aiding CEGAR algorithms in their search for abstract semantics [Grigore and Yang 2016], as well as help tune abstract interpreter heuristics [Cha et al. 2016; Chae et al. 2017; He et al. 2020; Heo et al. 2016, 2017; Jeong et al. 2017; Oh et al. 2015; Singh et al. 2018], (ii) directly compute candidate program invariants or specifications [Beckman and Nori 2011; Kremenek et al. 2006; Livshits et al. 2009; Si et al. 2018], (iii) rank the list of bugs reported by a program analysis tool, in order of the probability of the bug being a true program bug (as opposed to being a false positive) [Kremenek et al. 2004; Kremenek and Engler 2003; Raghothaman et al. 2018] and to allow the use of developer provided feedback in order to update the list of reported bugs [Kremenek et al. 2004; Mangal et al. 2015], (iv) guide the tactics to be used by proof search algorithms [Alemi et al. 2016; Bansal et al. 2019; Blaauwbroek et al. 2020; Chen et al. 2019; Kaliszyk et al. 2018; Loos et al. 2017; Sanchez-Stern et al. 2020], (v) infer the likely types or annotations of a program [Allamanis et al. 2020; Bielik et al. 2017; Hellendoorn et al. 2018; Wei et al. 2019], or predict program behaviors [Allamanis et al. 2018; Alon et al. 2018; Raychev et al. 2015].

Aiding static reasoning via dynamic checks. The use of dynamic checks as a mechanism to help with static reasoning about programs has been a topic of intense investigation in recent years, particularly in the context of gradual typing [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006]. Gradual typing aims to reason about programs written in a mixture of typing disciplines, and employs dynamic checks, wherever necessary, to translate between the different typing discplines. However, the idea of dynamic checks as an aid for type-based reasoning [Abadi et al. 1991; Cartwright and Fagan 1991; Flanagan 2006; Furr et al. 2009; Henglein 1994; Knowles and Flanagan 2010; Thatte 1989] and for computing more precise invariants [Bastani et al. 2015, 2019; Devecsery et al. 2018; Stulova et al. 2016] has been repeatedly used over the last thirty years. In the opposite direction, starting from programs already embedded with dynamic checks or constracts, static reasoning has been used to remove the dynamic checks, if possible and reduce the run time

overhead [Choi et al. 2002; Elmas et al. 2007; Myers 1999; Nagarakatte et al. 2010; Necula et al. 2002; Nguyen et al. 2017, 17ed, 2014; Rhodes et al. 2017; Tobin-Hochstadt and Van Horn 2012].

6 CONCLUSION

We study the proof strategies employed by algorithms that search for proofs of program judgments. We are particularly interested in three broad strategies, namely, computing semantic program invariants, reasoning hypothetically about programs by embedding them with dynamic/run time checks, and using data representing observations about program behavior to help reason about a program. We present a meta-theoretic construction, referred as observational abstract interpreter, that combines these three reasoning strategies. An observational abstract interpreter uses program observations to infer hypotheses about program behavior, and computes hypothetical semantic invariants of the program. These hypotheses are embedded in the program as dynamic checks. Our design of observational abstract interpreters is heavily inspired by the abstracting abstract machines methodology of Van Horn and Might for constructing concrete and abstract interpreters of higher-order languages, and the monadically refactored design of these interpreters. We formalize our ideas in the context of a simple higher-order language (λ_S) with built-in integers. We construct an observational interpreter for interval analysis of λ_S programs.

ACKNOWLEDGMENTS

I am grateful to Alessandro Orso, David Devecsery, and David Darais for the discussions that led to this work.

REFERENCES

Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. 1991. Dynamic Typing in a Statically Typed Language. *ACM Transactions on Programming Languages and Systems* 13, 2 (April 1991), 237–268.

Alexander A. Alemi, François Chollet, Niklas Een, Geoffrey Irving, Christian Szegedy, and Josef Urban. 2016. DeepMath - Deep Sequence Models for Premise Selection. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., Barcelona, Spain, 2243–2251.

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* 51, 4 (July 2018), 81:1–81:37.

Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, London, UK, 91–105.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. *ACM SIGPLAN Notices* 53, 4 (June 2018), 404–419.

Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. Association for Computing Machinery, Portland, Oregon, 4–16.

Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic Inference of Static Types for Ruby. *ACM SIGPLAN Notices* 46, 1 (Jan. 2011), 459–472.

Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, Cham, 25–46.

Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *International Conference on Machine Learning*. 454–463.

Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Interactively Verifying Absence of Explicit Information Flows in Android Apps. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. Association for Computing Machinery, Pittsburgh, PA, USA, 299–315.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active Learning of Points-to Specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, Philadelphia, PA, USA, 678–692.

Osbert Bastani, Rahul Sharma, Lazaro Clapp, Saswat Anand, and Alex Aiken. 2019. Eventually Sound Points-To Analysis with Specifications. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:28.

Nels E. Beckman and Aditya V. Nori. 2011. Probabilistic, Modular and Scalable Inference of Typestate Specifications. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, San Jose, California, USA, 211–221.

Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons, Sai Deep Tetali, and Aditya V. Thakur. 2010. Proofs from Tests. *IEEE Transactions on Software Engineering* 36, 4 (July 2010), 495–508.

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2017. Learning a Static Analyzer from Data. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 233–253.

Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. Tactic Learning and Proving for the Coq Proof Assistant. In *EPiC Series in Computing*, Vol. 73. EasyChair, 138–150.

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. Association for Computing Machinery, Waikiki, Honolulu, HI, USA, 241–250.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (Dec. 2011), 26:1–26:66.

Robert Cartwright and Mike Fagan. 1991. Soft Typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. Association for Computing Machinery, Toronto, Ontario, Canada, 278–292.

Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. 2016. Learning a Strategy for Choosing Widening Thresholds from a Large Codebase. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Atsushi Igarashi (Ed.). Springer International Publishing, Cham, 25–41.

Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically Generating Features for Learning Program Analysis Heuristics for C-like Languages. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 101:1–101:25.

Jia Chen, Jiayi Wei, Yu Feng, Osbert Bastani, and Isil Dillig. 2019. Relational Verification Using Reinforcement Learning. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 141:1–141:30.

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. *ACM SIGPLAN Notices* 37, 5 (May 2002), 258–269.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *J. ACM* 50, 5 (Sept. 2003), 752–794.

Patrick Cousot. 1999. The calculational design of a generic abstract interpreter. *Calculational system design* (1999), 421–505.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. Association for Computing Machinery, Los Angeles, California, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. Association for Computing Machinery, San Antonio, Texas, 269–282.

Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation* 2, 4 (Aug. 1992), 511–547.

Christoph Csallner, Yannis Smaragdakis, and Tao Xie. 2008. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. *ACM Transactions on Software Engineering and Methodology* 17, 2 (May 2008), 8:1–8:37.

David Darais and David Van Horn. 2019/ed. Constructive Galois Connections. *Journal of Functional Programming* 29 (2019/ed).

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (Aug. 2017), 12:1–12:25.

David Darais, Matthew Might, and David Van Horn. 2015. Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. Association for Computing Machinery, Pittsburgh, PA, USA, 552–571.

David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2018. Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing

Machinery, Williamsburg, VA, USA, 348–362.

Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated Error Diagnosis Using Abductive Inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. Association for Computing Machinery, Beijing, China, 181–192.

Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended Analysis for Performance Understanding of Framework-Based Applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. Association for Computing Machinery, London, United Kingdom, 118–128.

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-Aware Java Runtime. *ACM SIGPLAN Notices* 42, 6 (June 2007), 245–255.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming* 69, 1-3 (Dec. 2007), 35–45.

Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. 2017. Sampling Invariants from Frequency Distributions. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. 100–107.

Mattias Felleisen and D. P. Friedman. 1987. A Calculus for Assignments in Higher-Order Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. Association for Computing Machinery, Munich, West Germany, 314.

Cormac Flanagan. 2006. Hybrid Type Checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. Association for Computing Machinery, Charleston, South Carolina, USA, 245–256.

Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. 2009. Profile-Guided Static Typing for Dynamic Scripting Languages. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. Association for Computing Machinery, Orlando, Florida, USA, 283–300.

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 69–87.

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, St. Petersburg, FL, USA, 499–512.

Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2017. Heaps Don't Lie: Countering Unsoundness with Heap Snapshots. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 68:1–68:27.

Radu Grigore and Hongseok Yang. 2016. Abstraction Refinement Guided by a Learnt Probabilistic Model. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, St. Petersburg, FL, USA, 485–498.

Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. 2009. From Tests to Proofs. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Stefan Kowalewski and Anna Philippou (Eds.). Springer, Berlin, Heidelberg, 262–276.

Rajiv Gupta, Mary Lou Soffa, and John Howard. 1997. Hybrid Slicing: Integrating Dynamic Information with Static Analysis. *ACM Transactions on Software Engineering and Methodology* 6, 4 (Oct. 1997), 370–397.

Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2020. Learning Fast and Precise Numerical Analysis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, London, UK, 1112–1127.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, Lake Buena Vista, FL, USA, 152–162.

Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. *Science of Computer Programming* 22, 3 (June 1994), 197–230.

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis. In *Static Analysis (Lecture Notes in Computer Science)*, Xavier Rival (Ed.). Springer, Berlin, Heidelberg, 237–256.

Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-Learning-Guided Selectively Unsound Static Analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 519–529.

Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-Driven Context-Sensitivity for Points-to Analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 100:1–100:28.

Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. 2018. Reinforcement Learning of Theorem Proving. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 8822–8833.

Sven Keidel and Sebastian Erdweg. 2019. Sound and Reusable Components for Abstract Interpretation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 176:1–176:28.

Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional Soundness Proofs of Abstract Interpreters. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 72:1–72:26.

Johannes Kinder and Dmitry Kravchenko. 2012. Alternating Control Flow Reconstruction. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer, Berlin, Heidelberg, 267–282.

Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *ACM Transactions on Programming Languages and Systems* 32, 2 (Feb. 2010), 6:1–6:34.

Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. 2004. Correlation Exploitation in Error Ranking. *ACM SIGSOFT Software Engineering Notes* 29, 6 (Oct. 2004), 83–93.

Ted Kremenek and Dawson Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Static Analysis (Lecture Notes in Computer Science)*, Radhia Cousot (Ed.). Springer, Berlin, Heidelberg, 295–315.

Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. 2006. From Uncertainty to Belief: Inferring the Specification Within. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Seattle, Washington, 161–176.

Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: Using Dynamic Analysis to Infer Program Invariants in Separation Logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, Phoenix, AZ, USA, 788–801.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, San Francisco, California, USA, 333–343.

Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: Specification Inference for Explicit Information Flow Problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, Dublin, Ireland, 75–86.

Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. 2017. Deep Network Guided Proof Search. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. 85–105.

Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A User-Guided Approach to Program Analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, Bergamo, Italy, 462–473.

Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, London, UK, 1–15.

Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. 2002. Improving Program Slicing with Dynamic Points-to Data. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '02/FSE-10)*. Association for Computing Machinery, Charleston, South Carolina, USA, 71–80.

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. Association for Computing Machinery, San Antonio, Texas, USA, 228–241.

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. Association for Computing Machinery, Toronto, Ontario, Canada, 31–40.

Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. 2012. Abstractions from Tests. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. Association for Computing Machinery, Philadelphia, PA, USA, 373–386.

George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-Safe Retrofitting of Legacy Code. *ACM SIGPLAN Notices* 37, 1 (Jan. 2002), 128–139.

Phúc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2017. Soft Contract Verification for Higher-Order Stateful Programs. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 51:1–51:30.

Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2017/ed. Higher Order Symbolic Execution for Contract Verification and Refutation*. *Journal of Functional Programming* 27 (2017/ed).

Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. *ACM SIGPLAN Notices* 49, 9 (Aug. 2014), 139–152.

Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation. *ACM SIGPLAN Notices* 50, 10 (Oct. 2015), 572–588.

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, Santa Barbara, CA, USA, 42–56.

Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-Guided Program Reasoning Using Bayesian Inference. *ACM SIGPLAN Notices* 53, 4 (June 2018), 722–735.

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". *ACM SIGPLAN Notices* 50, 1 (Jan. 2015), 111–124.

Thomas Reps, Mooly Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Bernhard Steffen and Giorgio Levi (Eds.). Springer, Berlin, Heidelberg, 252–266.

John C. Reynolds. 2000. The Meaning of Types From Intrinsic to Extrinsic Semantics. *BRICS Report Series* 32 (June 2000).

Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. 2017. BigFoot: Static Check Placement for Dynamic Race Detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, Barcelona, Spain, 141–156.

H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating Correctness Proofs with Neural Networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2020)*. Association for Computing Machinery, London, UK, 1–10.

Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. 2008. Dynamic Inference of Likely Data Preconditions over Predicates by Tree Learning. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. Association for Computing Machinery, Seattle, WA, USA, 295–306.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. *ACM SIGPLAN Notices* 48, 6 (June 2013), 399–410.

Rahul Sharma and Alex Aiken. 2016. From Invariant Checking to Invariant Inference Using Randomized Search. *Formal Methods in System Design* 48, 3 (June 2016), 235–256.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013b. A Data Driven Approach for Algebraic Loop Invariants. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, Berlin, Heidelberg, 574–592.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. 2013a. Verification as Learning Geometric Concepts. In *Static Analysis (Lecture Notes in Computer Science)*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer, Berlin, Heidelberg, 388–411.

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 7751–7762.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *In Scheme and Functional Programming Workshop*.

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 211–229.

Nataliia Stulova, José F. Morales, and Manuel V. Hermenegildo. 2016. Reducing the Overhead of Assertion Run-Time Checks via Static Analysis. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP '16)*. Association for Computing Machinery, Edinburgh, United Kingdom, 90–103.

A. Thakur, A. Lal, J. Lim, and T. Reps. 2015. PostHat and All That: Automating Abstract Interpretation. *Electronic Notes in Theoretical Computer Science* 311 (Feb. 2015), 15–32.

Satish Thatte. 1989. Quasi-Static Typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. Association for Computing Machinery, San Francisco, California, USA, 367–381.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. Association for Computing Machinery, Portland, Oregon, USA, 964–974.

Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-Order Symbolic Execution via Contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. Association for Computing Machinery, Tucson, Arizona, USA, 537–554.

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. Association for Computing Machinery, Baltimore,

Maryland, USA, 51–62.

Niki Vazou, Éric Tanter, and David Van Horn. 2018. Gradual Liquid Type Inference. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 132:1–132:25.

Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. Association for Computing Machinery, Albuquerque, New Mexico, USA, 1–14.

P. Wadler and S. Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. Association for Computing Machinery, Austin, Texas, USA, 60–76.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2019. LambdaNet: Probabilistic Type Inference Using Graph Neural Networks. In *International Conference on Learning Representations*.

Shiyi Wei and Barbara G. Ryder. 2013. Practical Blended Taint Analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. Association for Computing Machinery, Lugano, Switzerland, 336–346.

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-Driven CHC Solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, Philadelphia, PA, USA, 707–721.

He Zhu, Aditya V. Nori, and Suresh Jagannathan. 2015. Learning Refinement Types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. Association for Computing Machinery, Vancouver, BC, Canada, 400–411.

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically Learning Shape Specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, Santa Barbara, CA, USA, 491–507.