Deriving Abstract Interpreters from Skeletal Semantics

Thomas Jensen
INRIA, Rennes
thomas.jensen@inria.fr

Vincent Rébiscoul
Université de Rennes, Rennes
vincent.rebiscoul@inria.fr

Alan Schmitt
INRIA, Rennes
alan.schmitt@inria.fr

This paper describes a methodology for defining an executable abstract interpreter from a formal description of the semantics of a programming language. Our approach is based on Skeletal Semantics and an abstract interpretation of its semantic meta-language. The correctness of the derived abstract interpretation can be established by compositionality provided that correctness properties of the core language-specific constructs are established. We illustrate the genericness of our method by defining a Value Analysis for a small imperative language based on its skeletal semantics.

1 Introduction

The derivation of provably correct static analyses from a formal specification of the semantics of a programming language is a long-standing challenge. The recent advances in the mechanisation of semantics has opened up novel perspectives for providing tool support for this task, thereby enabling the scaling of this approach to larger programming languages. This paper presents one such approach for mechanically constructing semantics-based program analysers from a formal description of the semantics of a programming language. We aim to provide methodologies which not only can prove the correctness of program abstractions but also lead to executable analysis techniques. Abstract Interpretation [4] has set out a methodology for defining an abstract semantics from an operational semantics and for proving a correctness relation between abstract and concrete semantics using Galois connections. The principle of abstract interpretation has been applied to a variety of semantic frameworks, including small-step and big-step (natural) operational semantics, and denotational semantics. An example of this methodology is to build an abstract semantics from a natural semantics [20]. Another example is Nielson's theory of abstract interpretation of two-level semantics [14] in which a semantic meta-language is equipped with binding-time annotations so that types and terms can be given a *static* and *dynamic* interpretation, leading to different but (logically) related interpretations.

In order for semantics-based program analysis to handle the complexity of today's programming languages, it is necessary to conceive a methodology that is built using some form of mechanised semantics. Examples of this include Verasco [8], a formally verified static analyser for the C programming language. It uses abstract interpretation techniques to perform value analyses, relational analyses… Verasco is written in Coq and the soundness of the analysis is guaranteed by a theorem: a program where the analysis does not raise an alarm is free of errors. Reasoning about program behaviours is possible as Verasco reuses the formalisation of the C semantics in Coq that was written for CompCert [11]. CompCert is a proved semantic preserving C compiler written in Coq.

Another example is the $\mathbb{K}$ [18] framework for writing semantics using rewriting rules. Rewriting rules make the formal definition of a semantics both flexible and relatively simple to write, and allows to mechanically derive objects from the semantics like an interpreter. However, this mechanization can be complex: $\mathbb{K}$-Java [2] is a formalization of Java in $\mathbb{K}$, with close to four hundred rewriting rules. It is unclear if it is possible to derive an analysis from a mechanization in $\mathbb{K}$.

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on
Expressiveness in Concurrency and Structural Operational Semantics
(EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 97–113, doi:10.4204/EPTCS.387.8

The key idea that we will pursue in this paper is that an abstract interpreter for a semantic meta-language combined with language-specific abstractions for a particular property yield a correct-by-construction abstract interpreter for the specific language and property. We describe how to obtain a correct program analyser for a programming language from its *skeletal* semantics. Skeletal Semantics [1] is a proposal for machine-representable semantics of programming languages.

The skeletal semantics of a language $\mathscr{L}$ is a partial description of the semantics of $\mathscr{L}$. Typically, a skeletal semantics will contain definitions of the constructs of the language and functions of evaluation of these constructs. A skeletal semantics is written in the meta-language Skel [17], a minimalist functional language. It is a *meta language* to describe the semantics of *object languages*. Skel has several semantics, called *interpretations*, (small step, big step [10], abstract interpretation), giving different semantics for the object languages.

Contributions

- We propose new interpretations of the semantic meta-language Skel that integrates the notion of program point in a systematic way.
- We define an abstract interpretation for Skel. The abstract interpretation of Skel combined with language-specific abstractions define an analyzer for the object language.
- We prove that the abstract interpretation of Skel is a sound approximation of the big-step interpretation of Skel, provided that some small language-dependent properties hold.
- We implement a program which, given a Skeletal Semantics, generates an executable abstract interpreter, and we test it on toy languages. We define a basic value analyzer for a small imperative language. A Control Flow Analysis for a λ-calculus is also presented in the long version of this paper [7].

2 Skeletal Semantics

Skeletal Semantics offers a framework to mechanise semantics of programming languages [1]. It uses a minimalist, functional, and strongly typed semantic meta-language called Skel [17], whose syntax is presented in Figure 1. The actual semantics of a language described in Skel is expressed by providing a (meta-)interpretation of the Skel language itself. In this paper, we will present two such interpretations: a big-step (or concrete) semantics and an abstract interpretation.

We illustrate Skel through the definition of the skeletal semantics of a toy imperative language called While. A Skeletal Semantics is a formal description of a language and consists of *declarations*. We start with some type declarations (production r_τ in Figure 1).

```
type ident
type lit
type store
type int
```

```
type expr =
| Const lit
| Var ident
| Plus (expr, expr)
| Leq(expr, expr)
| Rand (lit, lit)
```

```
type stmt =
| Skip
| Assign (ident, expr)
| Seq (stmt, stmt)
| If (expr, stmt, stmt)
| While (expr, stmt)
```

For While, there are four *unspecified* types (identifiers, literals, stores, integers) and two *specified* types (expressions and statements). Unspecified types is an useful trait of Skel, their definitions are unconstrained and they can be instantiated depending on the semantics of the object language being defined.

$$
\begin{array}{rrcl}
\textsc{Term} & t & ::= & x \mid C\,t \mid (t,..,t) \mid \lambda p : \tau \to S \\
\textsc{Skeleton} & S & ::= & t \mid t_0\; t_1..t_n \mid \textbf{let}\; p = S\; \textbf{in}\; S \mid \textbf{branch}\; S\,\text{or}\,..\,\text{or}\,S\; \textbf{end} \mid \\
 & & & \textbf{match}\; t\; \textbf{with}\; p \to S..p \to S\; \textbf{end} \\
\textsc{Pattern} & p & ::= & x \mid _ \mid C\; p \mid (p,..,p) \\
\textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau,..,\tau) \\
\textsc{Term decl} & r_t & ::= & \textbf{val}\; x : \tau \mid \textbf{val}\; x : \tau = t \\
\textsc{Type decl} & r_\tau & ::= & \textbf{type}\; b \mid \textbf{type}\; b = \text{"|"}\; C_1\; \tau_1..\text{"|"}\; C_n\; \tau_n \\
\textsc{Skeletal Semantics} & \mathscr{S} & ::= & (r_t|r_\tau)^*
\end{array}
$$

Figure 1: The Syntax of Skeletal Semantics

The specification of the integer type can be different for a big-step semantics or for an abstract interpretation. The `expr` and `stmt` types define expressions and statements of While programs. An expression can be a constant, a variable, an addition, a comparison, or a random integer. A statement can be a skip (an instruction that does nothing), an assignment, a sequence, a condition, or a loop. In addition to these declared types, one may build arrow types and tuple types.

We now turn to Skel's *term declarations* (production r_t of Figure 1), which may also be unspecified or specified. Unspecified terms are typically used for operations on values of unspecified types. For our While language, they are as follows.

```
val litToInt : lit → int
val add : (int,  int) → int
val lt: (int,  int) → int
val rand : (lit, lit) → int
val isZero: int → ()
val isNotZero: int → ()
val read : (ident, store) → int
val write : (ident, store, int) → store
```

The types for `isZero` and `isNotZero` may be surprising. These partial functions act as filters when used in branches, as detailed below.

Specified terms, on the other hand, are signatures associated with a *term*. A term is either a skeletal variable, a constructor applied to a term, a tuple, or an abstraction. The body of an abstraction is a skeleton, described below. Consider the declaration of term `eval_expr`.

```
val eval_expr ((s, e): (store, expr)): int =
  match e with
  | Const i → litToInt i
  | Var x → read (x, s)
  | Plus (e1, e2) →
     let v1 = eval_expr (s, e1) in
     let v2 = eval_expr (s, e2) in
     add (v1, v2)
  | Leq (e1, e2) →
     let v1 = eval_expr (s, e1) in
     let v2 = eval_expr (s, e2) in
     lt (v1, v2)
  | Rand (i1, i2) → rand (i1, i2)
  end
```

The first line is syntactic sugar for

```
val eval_expr: (store, expr) -> int = λ (s, e): (store, expr) ->
```

where the remainder of the description is the body of the abstraction. This body is a skeleton. A skeleton may be a term, an n-ary application, a let binding, a branching (detailed below), or a match. Here the

skeleton is a match, distinguishing between the different expressions which may be evaluated. For a constant expression, we call the unspecified term `litToInt` to convert the literal to an integer. For a variable, we read its value in the store. For an addition, we sequence the recursive evaluation of each subterm using a let binding, and we then apply the unspecified `add` term to perform the actual addition. Note that specified term and type declarations are all mutually recursive. The rest of the code does not use any additional feature.

We now turn to the second specified term declaration, to evaluate statements.

```
val eval_stmt ((s, t): (store, stmt)): store  =
  match t with
  | Skip → s
  | Assign (x, e) →
     let w = eval_expr (s, e) in
     write (x, s, w)
  | Seq (t1, t2) →
     let s' = eval_stmt (s, t1) in
     eval_stmt (s', t2)
  | If (cond, true, false) →
     let i = eval_expr (s, cond) in
     branch
       let () = isNotZero i in
       eval_stmt (s, true)
     or
       let () = isZero i in
       eval_stmt (s, false)
     end
  | While (cond, t') →
     let i = eval_expr (s, cond) in
     branch
       let () = isNotZero i in
       let s' = eval_stmt (s, t') in
       eval_stmt (s', t)
     or
       let () = isZero i in s
     end
  end
```

The code for the conditional and the loop illustrates the last feature of the language, branching. Branches are introduced with the `branch` keyword and are separated with the `or` keyword. They correspond to a form of a non-deterministic choice. Intuitively, in a big-step interpretation, any branch that succeeds may be taken. Branches may fail if a pattern matching in a let binding fails, or if the application of a term fails. For instance, the instantiation of the term `isNotZero` will not be defined on 0, making the whole branch fail when given 0 as argument. This is how we decide which branch to execute next for the conditional, and whether to loop in the `While` case.

3 Big-step Semantics of Skel

We give meaning to a Skeletal Semantics by providing *interpretations* of Skel. We first define the concrete, big-step semantics of Skel. Let $\mathscr{S}$ be an arbitrary Skeletal Semantics. We write $\mathrm{Funs}(\mathscr{S})$ for the set of pairs $(\Gamma, \lambda p : \tau_1 \to S_0)$ such that $\lambda p : \tau_1 \to S_0$ appears in Skeletal Semantics $\mathscr{S}$. The typing environment Γ gives types to the free variables of $\lambda p : \tau_1 \to S_0$. Full formal details are available in [7].

3.1 From Types to Concrete Values

The definitions of the sets of semantic values are presented on Figure 2a. They are defined by induction on the type. For each type τ, we write $V(\tau)$ the set of values of type τ.

A value with tuple type is a tuple of concrete values. A value of a specified type is a constructor applied to a value. A value with arrow type is a function that can either be a *named closure* or an *anonymous closure*. The set of named closure $\mathrm{NC}(\tau_1 \to \tau_2)$ and the set of anonymous closures $\mathrm{AC}(\tau_1 \to \tau_2)$ are defined on Figure 2b. A named closure denotes a function that is specified in the Skeletal Semantics

$$V(\tau_1 \times .. \times \tau_n) = V(\tau_1) \times .. \times V(\tau_n)$$
$$V(\tau_2) = \{\, C\ v \,|\, C : (\tau_1, \tau_2) \wedge v \in V(\tau_1)\,\}$$
$$V(\tau_1 \to \tau_2) = \mathrm{NC}(\tau_1 \to \tau_2) \cup \mathrm{AC}(\tau_1 \to \tau_2)$$

(a) Concrete values associated to each type

$$\mathrm{NC}(\tau_1 \to \tau_2) = \left\{ (f,n) \,\middle|\, \begin{array}{c} \textbf{val}\ f : \tau_1 \to \tau_2 [= t] \in \mathscr{S} \\ \mathrm{arity}(f) = n \end{array} \right\}$$
$$\mathrm{AC}(\tau_1 \to \tau_2) = \left\{ (\Gamma, p, S, E) \,\middle|\, \begin{array}{c} (\Gamma, \lambda p : \tau_1 \to S) \in \mathrm{Funs}(\mathscr{S}) \\ \Gamma \vdash E \\ \Gamma + p \leftarrow \tau_1 \vdash S : \tau_2 \end{array} \right\}$$

(b) Named Closures and Anonymous Closures

Figure 2: Definition of Concrete Values

$\mathscr{S}$, it is a pair of the name of the function and its arity. An anonymous closure is a tuple of a typing environment Γ, a pattern p to bind the argument upon application, a skeleton S which is the body of the function, and an environment E captured at the creation of the closure. An environment is a partial function mapping skeletal variable to concrete values. It is said to be consistent with typing environment Γ, written $\Gamma \vdash E$, if they have the same domain and if, for every $x \in \mathrm{dom}(\Gamma)$, we have $E(x) \in V^\sharp(\Gamma(x))$.

The unspecified types of a skeletal semantics must be instantiated to obtain an interpretation. In the case of While, the unspecified types are `ident`, `lit`, `int`, and `store`. They are instantiated as follows.

$$V(\texttt{store}) = \{\, s \,|\, s \in \mathscr{X} \hookrightarrow \mathbb{Z}\,\} \qquad V(\texttt{ident}) = \{\, x \,|\, x \in \mathscr{X}\,\} \qquad \text{with } \mathscr{X} = \{\, x, y, z, ..\}$$
$$V(\texttt{lit}) = \{\, l \,|\, l \in \mathbb{Z}\,\} \qquad V(\texttt{int}) = \{\, i \,|\, i \in \mathbb{Z}\,\}$$

Identifiers are taken from a countable set $\mathscr{X}$, literals and integers are relative integers, and stores are partial maps from identifiers to integers.

3.2 Interpretation of Unspecified Terms

In the following, we write $\mathrm{na}(\tau)$ when τ is not an arrow type. Take an unspecified term **val** $t : \tau_1 \to .. \to \tau_n \to \tau$ such that $\mathrm{na}(\tau)$, then an instantiation of t, written $[\![t]\!]$, is a function such that $[\![t]\!] \in (V(\tau_1) \times .. \times V(\tau_n)) \to \mathscr{P}_{fin}(V(\tau))$, where $\mathscr{P}_{fin}(X)$ is the set of finite subsets of X. In particular, if **val** $t : \tau$ and $\mathrm{na}(\tau)$, then $[\![t]\!] \subseteq V(\tau)$. Allowing the specification of a term to be a function returning a set is useful to model non-determinism.

We instantiate the unspecified functions of our While language. The expression (b) **?** e_1 **:** e_2 evaluates

$$\frac{E, S_1 \Downarrow v \qquad \vdash E + p \leftarrow v \rightsquigarrow E' \qquad E', S_2 \Downarrow w}{E, \text{let}\, p = S_1 \,\text{in}\, S_2 \Downarrow w}\ \textsc{LetIn} \qquad \frac{E, S_i \Downarrow v}{E, (S_1, .., S_n) \Downarrow v}\ \textsc{Branch}$$

Figure 3: Examples of Rules of the Big-Step Semantics

$$\frac{}{\vdash E + _ \leftarrow v \rightsquigarrow E}\ \textsc{wild} \qquad \frac{}{\vdash E + x \leftarrow v \rightsquigarrow \{x \mapsto v\}E}\ \textsc{var} \qquad \frac{\vdash E + p \leftarrow v \rightsquigarrow E'}{\vdash E + C\,p \leftarrow C\,v \rightsquigarrow E'}\ \textsc{const}$$

$$\frac{\vdash E + p_1 \leftarrow v_1 \rightsquigarrow E_2 \qquad .. \qquad \vdash E_n + p_n \leftarrow v_n \rightsquigarrow E'}{\vdash E + (p_1, .., p_n) \leftarrow (v_1, .., v_n) \rightsquigarrow E'}\ \textsc{tuple}$$

Figure 4: Rule of Extension of Environment using Pattern Matching

to e_1 is the condition b is true. Otherwise, it evaluates to e_2.

$$\begin{array}{rlrl}
[\![litToInt]\!](n) = \{n\} & & [\![add]\!](n_1, n_2) = \{n_1 + n_2\} \\
[\![lt]\!](n_1, n_2) = (n_1 < n_2)\ \textbf{?}\ \{1\}\ \textbf{:}\ \{0\} & & [\![rand]\!](n_1, n_2) = \{n \mid n_1 \leq n \leq n_2\} \\
[\![isZero]\!](n) = (n = 0)\ \textbf{?}\ \{()\}\ \textbf{:}\ \{\} & & [\![isNotZero]\!](n) = (n \neq 0)\ \textbf{?}\ \{()\}\ \textbf{:}\ \{\} \\
[\![read]\!](x, s) = \{s(x)\} & & [\![write]\!](x, s, n) = \{s\{x \mapsto n\}\}
\end{array}$$

The `rand` instantiation returns a set of values to capture the non-determinism of the instruction. The `isZero` function is defined only on input 0, whereas `isNotZero` is defined for all inputs except 0.

3.3 Big-step Semantics

We briefly present the big-step semantics of Skel: $E, S \Downarrow v$ is a relation from a skeletal environment E, mapping skeletal variables to values, and a skeleton S to a value v. The relation is defined by induction on S and is very similar to the natural semantics of λ-calculus with environment. We focus on two of the most important rules on Figure 3. The LetIn rule evaluates a let-binding by first evaluating S_1, next binding the result to the pattern in the current environment, then finally evaluating S_2 in the extended environment. The Branch rule describes how to evaluate a branching: any branch that successfully reduces to a value may be taken. Finally, the pattern matching rules for environment extension are given in Figure 4. The whole set of rules is given in [7].

The big-step semantics of a branching explains the types of some unspecified terms seen earlier. The partiality of the instantiations of `isZero` and `isNotZero` functions are used in the semantics of While to prevent some branches to be taken. They act as filters: if a branch does not have a derivation because its filter is undefined on the input, the alternative is to take another branch.

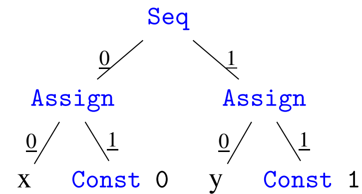
4 Big-step Semantics with Program Points

In this section, we introduce our first contribution, which is the integration of the notion of *program point* into the framework of Skeletal Semantics. A program point maps-to a precise fragment of a given

program. They play an important role in semantics-based program analysis, to indicate places where information about the execution is collected. Program points are essential to abstract interpretation as an abstract interpretation usually computes an abstraction of the state of the execution of the analysed program for each program point. Our formalisation of program points for Skeletal Semantics is modular and works for the big-step semantics of Skel, but also for the abstract interpretation of Skel, presented in Section 5.

In Skel, programs are values of an algebraic data type (ADT), such as `stmt` or `expr` in the While example. For instance, the skeletal term `Seq(Assign(x, Const 0), Assign(y, Const 1))` is a While **program** of type `stmt`. A program point is a path in the ADT of the program, encoded as a list of integers (underlined to distinguish them from natural numbers).

For example, ε is the empty path, it corresponds to the whole program. The path $\underline{01}$ corresponds to `Const 0`. The set of program points is thus $\mathtt{ppt} = \mathbb{N}^*$.



Let **prg** be a term of an ADT and pp a program point. We note **prg**@ pp the subterm of **prg** at program point pp. Formally, it is defined as follows.

$$v@\underline{\varepsilon} = v \qquad \mathtt{C}(v_0,..,v_{n-1})@(\underline{i}\,\mathrm{pp}) = v_i@\,\mathrm{pp} \quad \text{when } 0 \leq i \leq n-1$$

4.1 Building Values with Program Points

Our approach is to replace the values that correspond to programs with program points. These program points correspond to a sub-program of a main program that is a parameter of the interpretation. The values that ought to be replaced by program points should be values representing fragments of the program being executed. Therefore, we call $\mathscr{T}$ the set of *program types*, i.e., types representing programs. For instance, for the While language, $\mathscr{T} = \{\mathtt{stmt}, \mathtt{expr}\}$. Moreover, the interpretation is parametrised by a program **prg**, which is a value of a type $\tau \in \mathscr{T}$. For the While language, that could be **prg** = `Seq(Assign(x, Const 0), Assign(y, Const 1))`. Therefore, the rules to build values are unchanged except for the values with program types. Values with program types are defined by the following equation, where $V(\tau)$ is defined in Figure 2a.

$$\tau \in \mathscr{T} \implies V^{\mathtt{ppt}}_{\mathbf{prg}}(\tau) = \{\,\mathrm{pp} \in \mathtt{ppt} \mid \mathbf{prg}@\,\mathrm{pp} \in V(\tau)\,\}$$

Therefore in our example, ε is now a value of type `stmt` denoting the value **prg**, and $\underline{0}$ denotes the value `Assign(x, Const 0)`.

For each unspecified term x, we assume given an interpretation $[\![x]\!]^{\mathtt{ppt}}$, which is identical to the concrete interpretation $[\![x]\!]$ where program terms are replaced by program points. The full definition of this interpretation can be found in the long version of this paper [7].

4.2 Pattern-matching of Program Points

Replacing some values with program points does not change the interpretation of Skel, except when matching a program point with a pattern. Indeed, a program point pp corresponds to the sub-program **prg**@ pp if it exists, and it might be matched against a pattern `C` p. To handle this case, the program point is *unfolded*, meaning the constructor at **prg**@ pp is revealed, and the parameters of the constructor are replaced with program points if their type is a program type. To give an example, given **prg** as before, unfolding ε gives `Seq`$(\underline{0},\underline{1})$: the constructor is revealed and the parameters are program points because

they both have type `stmt` $\in \mathscr{T}$. On the other hand, unfolding $\underline{00}$ directly returns `x`, as identifiers are not program types in this example. This unfolding mechanism is added to pattern matching via the following rule:

$$\frac{\begin{array}{c}\mathbf{prg}@\,\mathrm{pp} = \mathtt{C}\,(v'_0, .., v'_{n-1}) \qquad \mathtt{C} : (\tau_0 \times .. \times \tau_{n-1}, \tau) \\ v_j = \mathbf{if}\ \tau_j \in \mathscr{T}\ \mathbf{then}\ \mathrm{pp}\,\underline{j}\ \mathbf{else}\ v'_j \qquad \mathscr{T}, \mathbf{prg} \vdash E + p \leftarrow (v_0, .., v_{n-1}) \rightsquigarrow E'\end{array}}{\mathscr{T}, \mathbf{prg} \vdash E + \mathtt{C}\,p \leftarrow \mathrm{pp} \rightsquigarrow E'}\ \text{UNFOLD}$$

Note that the pattern matching is now parametrised with $\mathscr{T}$ and **prg**. To perform the pattern-matching of pp with $\mathtt{C}\ p$, the value $\mathbf{prg}@$ pp must have constructor `C` at the root. The parameters that have a program type are replaced by their program point and the pattern-matching is performed recursively.

5 Abstract Interpretation of Skel

We present our main contribution in this paper, which is the definition of an abstract interpreter of Skel that is sound with respect to the big-step semantics of Section 4. This abstract interpreter will serve as the foundations of a methodology for building abstract interpreters for object languages from their skeletal semantics. In this methodology, several ingredients must be provided to generate such an abstract interpreter.

- An abstract instantiation of unspecified types to sets of abstract values, each of which comes with a concretisation function, a partial order, and an abstract union operator. Our framework automatically extends these definitions to all types.
- A *state of the abstract interpretation* (AI-state in the following) used to carry additional information during the abstract evaluation. For instance, in our While language, the AI-state records the current approximation of the store for each program point.
- The user may give functions to update the AI-state at the start and end of a call to a specified function. This is typically used for evaluation functions, such as `eval_stmt`, to record information right before and right after executing a sub-program.
- An instantiation of the unspecified terms, based on the definition of the abstract instantiation of types and the AI-state.

Our framework provides an abstract meta-semantics of Skel that threads the AI-state through the evaluation, including calls to unspecified terms. As the foundational correctness property of the methodology, we prove that if the abstract instantiation of types and terms provided by the user satisfy some correctness criteria, then the whole abstract interpreter that is generated is also correct.

5.1 Abstract Values

Abstract values are built similarly to concrete ones, based on their types, and we write $V^\sharp(\tau)$ for the set of abstract values of type τ. We first define the sets of abstract (named) closures at the top of Figure 5. Abstract named closures are identical to concrete named closures: they are pairs of a name of a function defined in the skeletal semantics and its arity. Abstract closures consist of a typing environment, a pattern, a skeleton, and an abstract environment that is consistent with the typing environment. An abstract environment $E^\sharp$ is a mapping from skeletal variables to abstract values. It is said to be consistent with typing environment Γ, written $\Gamma \vdash E^\sharp$, if they have the same domain and if, for every $x \in \mathrm{dom}(\Gamma)$, we have $E^\sharp(x) \in V^\sharp(\Gamma(x))$.

$$\mathrm{NC}(\tau_1 \to \tau_2) = \left\{ (f,n) \,\middle|\, \begin{array}{c} \textbf{val}\ f : \tau_1 \to \tau_2[=t] \in \mathscr{S} \\ \mathrm{arity}(f) = n \end{array} \right\}$$
$$\mathrm{AC}(\tau_1 \to \tau_2) = \left\{ (\Gamma, p, S, E^\sharp) \,\middle|\, \begin{array}{c} (\Gamma, \lambda p : \tau_1 \to S) \in \mathrm{Funs}(\mathscr{S}) \\ \Gamma \vdash E^\sharp \\ \Gamma + p \leftarrow \tau_1 \vdash S : \tau_2 \end{array} \right\}$$
$$V^\sharp(\tau_1 \times .. \times \tau_n) = \mathscr{P}_{fin}(V^{\sharp\star}(\tau_1) \times .. \times V^{\sharp\star}(\tau_n))$$
$$V^\sharp(\tau_2) = \{C\ v^\sharp \mid C : (\tau_1, \tau_2) \wedge v^\sharp \in V^{\sharp\star}(\tau_1)\} \cup \{\bot_{\tau_2}, \top_{\tau_2}\}$$
$$V^\sharp(\tau_1 \to \tau_2) = \mathscr{P}(\mathrm{NC}(\tau_1 \to \tau_2)) \cup \mathscr{P}(\mathrm{AC}(\tau_1 \to \tau_2))$$

Figure 5: Abstract Values for Specified Types

We assume that a partial order on $V^\sharp(\tau)$ is provided for each unspecified type τ, and that it includes a smallest value, denoted by $\bot_\tau$, and a largest value, denoted by $\top_\tau$. In the case of While, we instantiate `ident` with the flat lattice of $\mathscr{X}$, `lit` with the flat lattice of integers, `int` with closed intervals of $\mathbb{Z} \cup \{-\infty, +\infty\}$, and `store` with a partial mapping from identifiers to non-empty intervals.

We define abstract values for specified types in Figure 5, writing $V^{\sharp\star}(\tau)$ for $V^\sharp(\tau) \setminus \{\bot_\tau\}$. Abstract tuples are finite sets of tuples of (non-bottom) abstract values, with $\bot_{\tau_1 \times .. \times \tau_n}$ being the empty set. We use sets to retain some precision in the analysis. Abstract values of an algebraic data type are simply constructors applied to an abstract value of the correct type. Finally, functional abstract values are sets of abstract closures of this type, with $\bot_{\tau_1 \to \tau_2}$ being the empty set.

The AI-state $\mathscr{A}$ contains information collected throughout the abstract interpretation. It is dependent on the analysis and the language, and therefore must be provided, similarly to unspecified values. Moreover, a partial order and a union must be given for abstract states. In the case of while, the AI-state records information about the abstract store before (`In`) and after (`Out`) every program point. We write `Pos` for either `In` or `Out`. We then define $\mathscr{A}$ as a mapping from program points and a `Pos` to abstract while stores. We have $\mathscr{A}_1 \sqsubseteq^\sharp \mathscr{A}_2$ if $\mathrm{dom}(\mathscr{A}_1) \subseteq \mathrm{dom}(\mathscr{A}_2)$ and for any $(\mathrm{pp}, \mathtt{Pos}) \in \mathrm{dom}(\mathscr{A}_1)$, we have $\mathscr{A}_1(\mathrm{pp}, \mathtt{Pos}) \sqsubseteq^\sharp_{store} \mathscr{A}_2(\mathrm{pp}, \mathtt{Pos})$. We define $\mathscr{A}_1 \sqcup^\sharp \mathscr{A}_2$ as the mapping from $\mathrm{dom}(\mathscr{A}_1) \cup \mathrm{dom}(\mathscr{A}_2)$ that relates $(\mathrm{pp}, \mathtt{Pos})$ to $\mathscr{A}_1(\mathrm{pp}, \mathtt{Pos}) {\sqcup^\sharp}_{int} \mathscr{A}_2(\mathrm{pp}, \mathtt{Pos})$.

5.2 Operations on Abstract Values

Each abstract domain has a partial order and an associated join operator. In addition, a concretisation function that returns a set of concrete values defines the meaning of each abstract value. All of these functions are indexed by types (or type environments when they deal with environments). We assume they are provided for non-specified types, and show in this section how to extend them to all types.

A concretisation function for type τ maps an abstraction state and an abstract value in $V^\sharp(\tau)$ to $\mathscr{P}(V(\tau))$, a set of concrete values. We also define a function of concretisation γ_Γ which maps abstract

skeletal environments to sets of concrete skeletal environments.

$$\gamma_{\tau_1\times..\times\tau_n}(\mathscr{A},t^\sharp) = \bigcup_{(v_1^\sharp,..,v_n^\sharp)\in t^\sharp} \gamma_{\tau_1}(\mathscr{A},v_1^\sharp)\times..\times\gamma_{\tau_n}(\mathscr{A},v_n^\sharp)$$

$$\gamma_{\tau_2}(\mathscr{A},C\,v^\sharp) = \left\{ C\,v \mid C:(\tau_1,\tau_2),\, v\in\gamma_{\tau_2}(\mathscr{A},v^\sharp) \right\}$$

$$\gamma_{\tau_1\to\tau_2}(\mathscr{A},F) = \{\,(f,n)\mid(f,n)\in F\,\}\cup\left\{ (\Gamma,p,S,E) \,\middle|\, (\Gamma,p,S,E^\sharp)\in F \wedge E\in\gamma_\Gamma(\mathscr{A},E^\sharp) \right\}$$

$$\gamma_\Gamma(\mathscr{A},E^\sharp) = \left\{ E \,\middle|\, \Gamma\vdash E\wedge\Gamma\vdash E^\sharp\wedge\forall x\in\mathrm{dom}(\Gamma), E(x)\in\gamma_{\Gamma(x)}(\mathscr{A},E^\sharp(x)) \right\}$$

$$\gamma(\mathscr{A},\bot_\tau)=\emptyset \qquad \gamma(\mathscr{A},\top_\tau)=V^\sharp(\tau)$$

In the case of While, the concretisation function for `ident` and `lit` are immediate as they are flat lattices. The concretisation function for an interval i is the set of integers it contains: $\gamma_{\mathtt{int}}(i)=\{\,n\mid n\in i\,\}$. Finally, the concretisation of an abstract store $\sigma^\sharp$ is

$$\gamma_{\mathtt{store}}(\sigma^\sharp) = \left\{ \sigma \,\middle|\, \mathrm{dom}(\sigma)=\mathrm{dom}(\sigma^\sharp)\wedge\forall x\in\mathrm{dom}(\sigma), \sigma(x)\in\gamma_{\mathtt{int}}(\sigma^\sharp(x)) \right\}$$

To compare abstract values, we define partial orders that are relations, but we call them functions as they can be viewed as boolean functions. For every unspecified type τ, we assume a comparison function $\sqsubseteq^\sharp_\tau$ which is a partial order between abstract values. It must satisfy the following property: for any value $v^\sharp\in V^\sharp(\tau)$, we have $\bot_\tau\sqsubseteq^\sharp_\tau v^\sharp$ and $v^\sharp\sqsubseteq^\sharp_\tau\top_\tau$. For every other type, the comparison function is the smallest partial order that satisfies the following equations.

$$C\,v^\sharp\sqsubseteq^\sharp_{\tau_a} C\,w^\sharp \iff v^\sharp\sqsubseteq^\sharp_\tau w^\sharp \text{ with } C:(\tau,\tau_a)$$

$$v^\sharp\sqsubseteq^\sharp_{\tau_1\times..\times\tau_n} w^\sharp \iff \forall(v_1^\sharp,..,v_n^\sharp)\in v^\sharp, \exists(w_1^\sharp,..,w_n^\sharp)\in w^\sharp \text{ such that } \forall i\in[1..n], v_i^\sharp\sqsubseteq^\sharp_{\tau_i} w_i^\sharp$$

$$F_1\sqsubseteq^\sharp_{\tau_1\to\tau_2} F_2 \iff \begin{cases} (f,n)\in F_1 \implies (f,n)\in F_2 \\ (\Gamma,p,S,E_1^\sharp)\in F_1 \implies \exists(\Gamma,p,S,E_2^\sharp)\in F_2, E_1^\sharp\sqsubseteq^\sharp_\Gamma E_2^\sharp \end{cases}$$

$$E_1^\sharp\sqsubseteq^\sharp_\Gamma E_2^\sharp \iff \Gamma\vdash E_1^\sharp\wedge\Gamma\vdash E_2^\sharp\wedge\forall x\in\mathrm{dom}(E_1^\sharp), E_1^\sharp(x)\sqsubseteq^\sharp_{\Gamma(x)} E_2^\sharp(x)$$

$$v^\sharp\sqsubseteq^\sharp_\tau\top_\tau \qquad \bot_\tau\sqsubseteq^\sharp_\tau v^\sharp$$

Most rules are straightforward. To compare two functions, all named closures of the left function must be in the right function. Moreover, for all closures in the left function, there must be a closure in the right function with the same pattern and skeleton, but with a bigger abstract environment. Abstract environments are compared using point-wise lifting. For our While language, we have $i_1\sqsubseteq^\sharp_{\mathtt{int}} i_2$ if the interval i_1 is included in i_2, and $\sigma_1^\sharp\sqsubseteq^\sharp_{\mathtt{store}}\sigma_2^\sharp$ if for all x in $\mathrm{dom}(\sigma_1^\sharp)$, we have $\sigma_1^\sharp(x)\sqsubseteq^\sharp_{\mathtt{int}}\sigma_2^\sharp(x)$.

Definition 1 *A concretion function γ_τ is* monotonic *iff for any $v_1^\sharp\sqsubseteq^\sharp_\tau v_2^\sharp$ and $\mathscr{A}_1\sqsubseteq^\sharp\mathscr{A}_2$. we have $\gamma_\tau(\mathscr{A}_1,v_1^\sharp)\subseteq\gamma_\tau(\mathscr{A}_2,v_2^\sharp)$.*

Lemma 1 *$\gamma_{\mathtt{ident}}$, $\gamma_{\mathtt{lit}}$, $\gamma_{\mathtt{int}}$ and $\gamma_{\mathtt{store}}$ are monotonic.*

For each type, an upper bound (or join) is defined. For every non-specified type τ, we assume an upper bound $\sqcup^\sharp{}_\tau$. The definition of $\sqcup^\sharp{}_{\mathtt{ident}}$ and $\sqcup^\sharp{}_{\mathtt{lit}}$ have the usual definition for flat lattices. $\sqcup^\sharp{}_{\mathtt{int}}$ is the convex hull of two intervals and $\sqcup^\sharp{}_{\mathtt{store}}$ is the usual point-wise lifting of the abstract union of

integers. Moreover, we note $\nabla^\sharp_{\texttt{int}}$ the widening on intervals (define below) and $\nabla^\sharp_{\texttt{store}}$ the point-wise lifting of the widening of intervals.

$$[n_1,n_2]\nabla^\sharp_{\texttt{int}}[m_1,m_2] = \left[\begin{cases} n_1 & \text{if } n_1 \leq m_1 \\ -\infty & \text{otherwise}\end{cases}, \begin{cases} n_2 & \text{if } m_2 \leq n_2 \\ +\infty & \text{otherwise}\end{cases}\right]$$

We extend it for every other type.

$$(C\ v^\sharp)\sqcup^\sharp{}_{\tau_2}(C\ w^\sharp) = C\ (v^\sharp\sqcup^\sharp{}_{\tau_1}w^\sharp) \text{ with } C:(\tau_1,\tau_2) \qquad E_1^\sharp\sqcup^\sharp{}_\Gamma E_2^\sharp = \left\{x\in\text{dom}(\Gamma)\mapsto E_1^\sharp(x)\sqcup^\sharp{}_{\Gamma(x)}E_2^\sharp(x)\right\}$$

$$(C\ v^\sharp)\sqcup^\sharp{}_{\tau_2}(D\ w^\sharp) = \top_{\tau_2} \text{ with } C:(\tau_1,\tau_2)\wedge D:(\tau_1',\tau_2) \qquad v^\sharp\sqcup^\sharp{}_\tau\top_\tau = \top_\tau\sqcup^\sharp{}_\tau v^\sharp = \top_\tau$$

$$v^\sharp\sqcup^\sharp{}_{\tau_1\times..\times\tau_n}w^\sharp = v^\sharp\cup w^\sharp \qquad v^\sharp\sqcup^\sharp{}_\tau\bot_\tau = \bot_\tau\sqcup^\sharp{}_\tau v^\sharp = v^\sharp$$

$$F_1\sqcup^\sharp{}_{\tau_1\to\tau_2}F_2 = F_1\cup F_2$$

Joining two algebraic values with the same constructor is joining their parameters, and joining algebraic values with different constructors yields top. The join of abstract tuples or abstract functions is their union. Joining abstract environments is done by point-wise lifting. For each type, top is an absorbing element, and bottom is the neutral element.

Lemma 2 $\sqsubseteq^\sharp_{\texttt{ident}}$, $\sqsubseteq^\sharp_{\texttt{lit}}$, $\sqsubseteq^\sharp_{\texttt{int}}$, $\sqsubseteq^\sharp_{\texttt{store}}$ *are orders.* $\sqcup^\sharp_{\texttt{ident}}$, $\sqcup^\sharp_{\texttt{lit}}$, $\sqcup^\sharp_{\texttt{int}}$, $\sqcup^\sharp_{\texttt{store}}$ *give an upper bound.*

Lemma 3 *If for all unspecified types τ_u, γ_{τ_u} is monotonic, then for all τ, γ_τ is also monotonic.*

Lemma 4 *If for every unspecified type τ_u, $\sqsubseteq^\sharp_{\tau_u}$ is an order and $\sqcup^\sharp{}_{\tau_u}$ gives an upper bound, then for all τ, $\sqsubseteq^\sharp_\tau$ is an order and $\sqcup^\sharp{}_\tau$ gives an upper bound.*

Finally, we give an abstract specification of unspecified terms. As an illustration, here are a few specifications from our running example.

$$[\![litToInt]\!]^\sharp(n) = [n,n] \qquad [\![add]\!]^\sharp([n_1,n_2],[m_1,m_2]) = [n_1+m_1,n_2+m_2]$$

$$[\![read]\!]^\sharp(x,s^\sharp) = s^\sharp(x) \qquad [\![write]\!]^\sharp(x,s,[n_1,n_2]) = s^\sharp\{x\mapsto[n_1,n_2]\}$$

Definition 2 *Let x be an unspecified term of type τ, such that* $\text{na}(\tau)$. *We say that $[\![x]\!]^\sharp$ is a* ***sound approximation*** *of $[\![x]\!]^{ppt}$ if and only if:*

$$\forall\mathscr{A},\ [\![x]\!]^{ppt}\subseteq\gamma(\mathscr{A},[\![x]\!]^\sharp)$$

Definition 3 *Let f be an unspecified term of type $\tau_1\to..\to\tau_n\to\tau$ where* $\text{na}(\tau)$. *We say that $[\![f]\!]^\sharp$ is a* ***sound approximation*** *of $[\![f]\!]^{ppt}$ iff $\forall v_i\in V^{ppt}_{\boldsymbol{prg}}(\tau_i)$, $\forall v_i^\sharp\in V^\sharp(\tau_i)$, and for all abstract state $\mathscr{A}$, if*

$$\left.\begin{array}{r} v_i\in\gamma_{\tau_i}(\mathscr{A},v_i^\sharp) \\ [\![f]\!]^\sharp(\mathscr{A},v_1^\sharp,..,v_n^\sharp) = \mathscr{A}',w^\sharp \end{array}\right\} \implies [\![f]\!]^{ppt}(v_1,..,v_n)\subseteq\gamma_\tau(\mathscr{A}',w^\sharp)$$

Lemma 5 *The abstract instantiations of the unspecified terms for While are sound approximation of the concrete instantiations of the unspecified terms.*

$$\frac{E^\sharp(x) = v^\sharp}{E^\sharp, x \Downarrow v^\sharp}\ \textsc{Var} \qquad \frac{\mathbf{val}\ f : \tau_1 \to .. \to \tau_n \to \tau[= t] \in \mathscr{S} \qquad \mathrm{na}(\tau)}{E^\sharp, f \Downarrow \{(f,n)\}}\ \textsc{TermClos}$$

$$\frac{\mathbf{val}\ x : \tau = t \in \mathscr{S} \qquad \emptyset, t \Downarrow v^\sharp \qquad \mathrm{na}(\tau)}{E^\sharp, x \Downarrow v^\sharp}\ \textsc{TermSpec} \qquad \frac{\mathbf{val}\ x : \tau \in \mathscr{S} \qquad \mathrm{na}(\tau)}{E^\sharp, x \Downarrow [\![x]\!]^\sharp}\ \textsc{TermUnspec}$$

$$\frac{E^\sharp, t \Downarrow v^\sharp}{E^\sharp, \mathrm{C}\,t \Downarrow \mathrm{C}\,v^\sharp}\ \textsc{Const} \qquad \frac{E^\sharp, t_1 \Downarrow v_1^\sharp \qquad .. \qquad E^\sharp, t_n \Downarrow v_n^\sharp}{E^\sharp, (t_1, .., t_n) \Downarrow \left\{ (v_1^\sharp, .., v_n^\sharp) \right\}}\ \textsc{Tuple}$$

$$\frac{}{\pi, E^\sharp, \lambda p : \tau \cdot S \Downarrow^\sharp \left\{ (p, S, E^\sharp) \right\}}\ \textsc{Clos} \qquad \frac{\pi, \mathscr{A}, E^\sharp, S_i \Downarrow^\sharp v_i^\sharp, \mathscr{A}_i}{\pi, \mathscr{A}, E^\sharp, (S_1..S_n) \Downarrow^\sharp \sqcup^\sharp v_i^\sharp, \sqcup^\sharp \mathscr{A}_i}\ \textsc{Branch}$$

$$\frac{\pi, \mathscr{A}, E^\sharp, S_1 \Downarrow^\sharp v^\sharp, \mathscr{A}' \qquad \mathscr{T}, \mathbf{prg} \vdash \left\{ E^\sharp \right\} + p \leftarrow v^\sharp \rightsquigarrow \left\{ E_1^\sharp, .., E_n^\sharp \right\} \qquad \pi, \mathscr{A}', E_i^\sharp, S_2 \Downarrow^\sharp w_i^\sharp, \mathscr{A}_i}{\pi, \mathscr{A}, E^\sharp, \mathrm{let}\, p = S_1 \,\mathrm{in}\, S_2 \Downarrow^\sharp \sqcup^\sharp w_i^\sharp, \sqcup^\sharp \mathscr{A}_i}\ \textsc{LetIn}$$

$$\frac{E^\sharp, t_i \Downarrow v_i^\sharp \qquad \pi, \mathscr{A}, v_0^\sharp\ v_1^\sharp..v_n^\sharp \Downarrow_{app} v^\sharp, \mathscr{A}'}{\pi, \mathscr{A}, E^\sharp, t_0\ t_1..t_n \Downarrow^\sharp v^\sharp, \mathscr{A}'}\ \textsc{App}$$

Figure 6: Abstract Interpretation of Skeletons and Terms

5.3 Abstract Interpretation of Skel

The abstract interpretation of skeletons is given on Figure 6. It maintains a callstack of specified function calls which is used to prevent infinite computations by detecting identical nested calls. A callstack is an ordered list of frames. The set of callstacks Π is defined as:

$$\frac{}{\mathbf{emp} \in \Pi}$$

$$\frac{\mathscr{A} \text{ an AI-state} \qquad \mathbf{val}\ f : \tau_1 \to .. \to \tau_n \to \tau\ =\ \mathtt{t} \in \mathscr{S} \qquad \mathrm{na}(\tau) \qquad v_i \in V^\sharp(\tau_i) \qquad \pi \in \Pi}{(f, \mathscr{A}, [v_1, .., v_n]) :: \pi \in \Pi}$$

The abstract interpretation of skeletons is similar to the big-step interpretation: the evaluation of terms is almost unchanged except that evaluating a closure or a tuple returns a singleton. When evaluating a skeleton (branch, let-binding, or application), a state of the abstract interpretation is carried through the computations.

In the BRANCH rule, all branches are evaluated and joined instead of only one branch being evaluated. Pattern matching now returns set of environments rather than a single one (explained later). As a consequence, the LETIN rule may evaluate S_2 in several abstract environments. This flexibility in the control-flow of the abstract interpreter allows us to do control flow analysis for λ-calculus. The APP rule evaluates all terms and pass a list of values to the application relation, defined in Figure 7.

Because the abstraction of a function is a set of closures and named closures, the APP-SET rule evaluates each one individually. The BASE rule returns the remaining value when all arguments have been

$$\frac{\begin{array}{c}\textbf{val}\ f : \tau_1 \to .. \to \tau_n \to \tau = t \in \mathscr{S} \\ \text{na}(\tau) \qquad \emptyset, t \Downarrow v^\sharp \qquad \textbf{update}^{in}_f(\mathscr{A}, [v^\sharp_1, .., v^\sharp_n]) = \mathscr{A}_1, [v'^\sharp_1, .., v'^\sharp_n] \qquad (f, \mathscr{A}_1, [v'^\sharp_1, .., v'^\sharp_n]) \notin \pi \\ (f, [v'^\sharp_1, .., v'^\sharp_n]) :: \pi, \mathscr{A}_1, v^\sharp\ v'^\sharp_1..v'^\sharp_n \Downarrow_{app} w^\sharp, \mathscr{A}_2 \qquad \textbf{update}^{out}_f(\mathscr{A}_2, [v'^\sharp_1, .., v'^\sharp_n], w^\sharp) = \mathscr{A}_3, w'^\sharp\end{array}}{\pi, \mathscr{A}, (f, n)\ v^\sharp_1..v^\sharp_n \Downarrow_{app} w'^\sharp, \mathscr{A}_3}\ \textsc{Spec}$$

$$\frac{\begin{array}{c}\textbf{val}\ f : \tau_1 \to .. \to \tau_n \to \tau = t \in \mathscr{S} \qquad \text{na}(\tau) \\ \emptyset, t \Downarrow v^\sharp \qquad \textbf{update}^{in}_f(\mathscr{A}, [v^\sharp_1, .., v^\sharp_n]) = \mathscr{A}_1, [v'^\sharp_1, .., v'^\sharp_n] \qquad (f, \mathscr{A}_1, [v'^\sharp_1, .., v'^\sharp_n]) \in \pi\end{array}}{\pi, \mathscr{A}, (f, n)\ v^\sharp_1..v^\sharp_n \Downarrow_{app} \bot, \mathscr{A}_1}\ \textsc{Spec-Loop}$$

$$\frac{\textbf{val}\ f : \tau_1 \to .. \to \tau_n \to \tau \in \mathscr{S} \qquad \text{na}(\tau) \qquad [\![f]\!]^\sharp(\mathscr{A}, v^\sharp_1, .., v^\sharp_n) = w^\sharp, \mathscr{A}'}{\pi, \mathscr{A}, (f, n)\ v^\sharp_1..v^\sharp_n \Downarrow_{app} w^\sharp, \mathscr{A}'}\ \textsc{Unspec}$$

Figure 7: Abstract Interpretation: Application

processed. The CLOS rule evaluates the body of the function S in all abstract environments returned by the matching of the pattern against the argument. The SPEC rule evaluates the call to a specified function and maintains invariants. Invariants depend on the analysis and the AI-state, therefore, language-dependent update functions can be provided to to maintain invariants before and after a call. The update functions must respect the following monotonicity constraints in order to ensure soundness:

Definition 4 *The update functions are said to be monotonic if and only if:*

$$\textbf{\textit{update}}^{in}_f(\mathscr{A}, [v^\sharp_1, .., v^\sharp_k]) = \mathscr{A}', [v'^\sharp_1, .., v'^\sharp_k] \implies \mathscr{A} \sqsubseteq^\sharp \mathscr{A}' \wedge (v^\sharp_1, .., v^\sharp_k) \sqsubseteq^\sharp (v'^\sharp_1, .., v'^\sharp_k)$$
$$\textbf{\textit{update}}^{out}_f(\mathscr{A}, [v^\sharp_1, .., v^\sharp_k], v^\sharp) = \mathscr{A}', v'^\sharp \implies \mathscr{A} \sqsubseteq^\sharp \mathscr{A}' \wedge v^\sharp \sqsubseteq^\sharp v'^\sharp$$

The update functions of While are defined as:

$$\textbf{update}^{in}_{\texttt{eval_stmt}}(\mathscr{A}, [s^\sharp_i, \text{pp}]) = \mathscr{A}\{(\text{pp}, \texttt{In}) \mapsto s^\sharp\}, [s^\sharp, \text{pp}] \qquad s^\sharp = s^\sharp_i \nabla^\sharp \mathscr{A}(\text{pp}, \texttt{In})$$
$$\textbf{update}^{out}_{\texttt{eval_stmt}}(\mathscr{A}, [s^\sharp_i, \text{pp}], s^\sharp_o) = \mathscr{A}\{(\text{pp}, \texttt{Out}) \mapsto s^\sharp\}, [s^\sharp, \text{pp}] \qquad s^\sharp = s^\sharp_o \sqcup^\sharp \mathscr{A}(\text{pp}, \texttt{Out})$$

The update functions maintain the AI-state which holds an input and an output abstract store for each program point. $\textbf{update}^{in}_{\texttt{eval_stmt}}(\mathscr{A}, [s^\sharp_i, \text{pp}])$ updates the input abstract store at program point pp for a greater abstract store, obtained by widening to ensure termination (discussed later), that contains $s^\sharp_i$, and the call to `eval_stmt` is done with this new abstract store. $\textbf{update}^{out}_{\texttt{eval_stmt}}(\mathscr{A}, [s^\sharp_i, \text{pp}], s^\sharp_o)$ makes a similar change to the AI-state for the output store. The update functions for `eval_expr` (not presented here) do not change the argument, the result or the AI-state.

Lemma 6 *The update functions for While previously defined are monotonic.*

The extension of environments, or pattern matching, is presented in [7] and now returns a set ξ of abstract environments as the abstraction of tuples is a finite set of tuples of abstract values. We thus return one abstract environment per tuple of abstract values in our abstract tuple.

The termination of our analysis is not formally proven. Our intuition is that an infinite derivation is necessarily caused by an infinity of calls to a specified function. Given a program point, widening the stores in the input update function for `eval_st` should ensure that the input store converges and that the SPEC-LOOP rule of the abstract interpretation ends the computation, as we have reached a local fixpoint for the program point.

5.4 Correctness of the Abstract Interpretation

Our methodology aims at defining mathematically correct abstract interpreters from Skeletal Semantics. In this section, we present a theorem stating that the abstract interpreter of Skel computes a correct approximation of the big-step semantics of Skel.

We state the following theorem of correctness that states that the abstract interpretation of Skel computes a sound approximation of the big-step interpretation of Skel.

Theorem 1 *Let $\mathscr{S}$ be a Skeletal Semantics with unspecified terms Te and unspecified types Ty, and let E and $E^\sharp$ be a concrete and abstract environment, respectively. Suppose*

- $\forall x \in Te$, *$[\![x]\!]^\sharp$ is a sound approximation of $[\![x]\!]^{ppt}$.*
- $\forall \tau \in Ty$, *γ_τ is monotonic.*
- ***update**^{in} and **update**^{out} are monotonic.*

Then:

$$\left.\begin{array}{c} E \in \gamma_\Gamma(\mathscr{A}_0, E^\sharp) \\ E, S \Downarrow v \\ \textbf{\textit{emp}}, \mathscr{A}_0, E^\sharp, S \Downarrow^\sharp v^\sharp, \mathscr{A} \end{array}\right\} \implies v \in \gamma(\mathscr{A}, v^\sharp)$$

Therefore, to prove the soundness of the analysis, it is sufficient to prove that the abstract instantiation of terms are sound approximation of the concrete ones, and that the update functions and concretisation functions are monotonic.

Let $\sigma_0 \in V^{\mathtt{ppt}}(\mathtt{store})$ and $\sigma_0^\sharp \in V^\sharp(\mathtt{store})$ be the concrete and abstract stores with empty domain. Let $E_0 = \{s \mapsto \sigma_0, t \mapsto \underline{\varepsilon}\}$ and $E_0^\sharp = \left\{s \mapsto \sigma_0^\sharp, t \mapsto \underline{\varepsilon}\right\}$ be a concrete and an abstract Skel environments. We recall that $\underline{\varepsilon}$ is the program point of the root of **prg**, the analysed program. Let $\mathscr{A}_0$ be the empty mapping from program points and flow tags (`In` or `Out`) to abstract stores.

Lemma 7 $\sigma_0 \in \gamma_{store}(\mathscr{A}_0, \sigma_0^\sharp)$

Lemma 8 *Let $\Gamma = \{s \mapsto \mathtt{store}, t \mapsto \mathtt{stmt}\}$, $E_0 \in \gamma_\Gamma(\mathscr{A}, E_0^\sharp)$.*

The abstract interpreter computes an abstract store that is a correct approximation of the concrete store returned by the big-step semantics.

Theorem 2

$$\left.\begin{array}{c} E_0, \mathtt{eval_stmt}\ (s,t) \Downarrow^{PP} \sigma \\ \textbf{\textit{emp}}, \mathscr{A}_0, E_0^\sharp, \mathtt{eval_stmt}\ (s,t) \Downarrow^\sharp \sigma^\sharp, \mathscr{A} \end{array}\right\} \implies \sigma \in \gamma(\mathscr{A}, \sigma^\sharp)$$

As an example, take **prg** $\equiv$ `x := 0; while (x < 3) x := x + 1`. The concrete and abstract interpretations will find that

$$E_0, \mathtt{eval_stmt}\ (s,t) \Downarrow^{PP} \{x \mapsto 3\}$$

$$\textbf{emp}, \mathscr{A}_0, E_0^\sharp, \mathtt{eval_stmt}\ (s,t) \Downarrow^\sharp \{x \mapsto [0, +\infty]\}, \mathscr{A}$$

In accordance with Theorem 2, we observe that $\{x \mapsto 3\} \in \gamma(\mathscr{A}, \{x \mapsto [0, +\infty]\})$
The abstract interpreter returns an imprecise result. Currently, our method fails to properly take into account the guards: the conditions of loops or conditional branchings are not used to refine the abstract values. In the previous While program, the guard of the loop is not used to get a precise abstract store in or after the loop. The skeletal semantics of the While language makes it unclear how to use the guards to modify the store, as it is syntactically the same before and after the evaluation of the condition.

The precision of the analysis also depends on the skeletal semantics of the language. An easy fix for our precision issue would be to modify the type of `isZero` and `isNotZero` functions such that they have type $(\texttt{store} \times \texttt{int}) \to \texttt{store}$. The abstract instantiations of these functions could then be used to refine the abstract stores.

6 Related Work

Our work is part of a large research effort to define sound analyses and build correct abstract interpreters from semantic description of languages. At its core, our approach is the Abstract Interpretation [4, 5] of a semantic meta-language. Abstract Interpretation is a method designed by Cousot and Cousot to define sound static analyses from a concrete semantics. In his Marktoberdorf lectures [3], Cousot describes a systematic way to derive an abstract interpretation of an imperative language from a concrete semantics and mathematically proved sound. We chose to define the Abstract Interpretation of Skel, as it is designed to mechanise semantics of languages. The benefit of analysing a meta-language is that a large part of the work to define and prove the correctness of the analysis is done once for every semantics mechanised with Skel. However, it is often less precise than defining a language specific abstract interpretation. Moreover, there have been several papers describing methods to derive abstract interpretation from different types of concrete semantics [4, 20, 13], we chose to derive abstract interpreters from a big-step semantics of Skel.

Schmidt [20] shows how to define an abstract interpretation for λ-calculus from a big-step semantics defined co-inductively. The abstract interpretation of Skel and its correctness proof follow the methods described in the paper. However Skel has more complex constructs than λ-calculus, especially branches. Moreover, the big-step of Skel is defined inductively, thus reasoning about non-terminating program is not possible. Also, to prove the correctness of the abstract interpretation of Skel, we relate the big-step derivation tree to the abstract derivation tree, similarly to Schmidt, but a key difference is that our proof is inductive when Schmidt's proof is co-inductive.

Lim and Reps propose the TSL system [12]: a tool to define machine-code instruction set and abstract interpretations. The specification of an instruction set in TSL is compiled into a Common Intermediate Representation (CIR). An abstract interpretation is defined on the CIR, therefore an abstract interpreter is derivable from any instruction set description. However, the TSL system is aimed at specifying and analysing machine code, and not languages in general. Moreover, it is unclear how it would be possible to define analyses on languages with more complex control-flow, like λ-calculus.

In the paper on Skeletal semantics, Bodin *et al.* [1] used skeletal semantics to relate concrete and abstract interpretations in order to prove correctness. An important difference between that work and the present is that their resulting abstract semantics is not computable, whereas our abstract interpretation can be executed as an analysis, as demonstrated by our implementation [19]. Moreover, our method computes an AI-state that collects information throughout the interpretation and allows to use widening using the update functions, rather than computing an Input/Output relation.

The idea of defining an abstract interpreter of a meta-language to define analyses for languages has been explored, for example by Keidel, Poulsen and Erdweg [9]. They use arrows [6] as meta-language to describe interpreters. The concrete and abstract interpreters share code using the unified interface of arrows. By instantiating language-dependent parts for the concrete interpretation and the abstract interpretation, they obtain two interpreters that can be proven sound compositionally by proving that the abstract language-dependent parts are sound approximation of the concrete language-dependent parts, similarly to Skel. However, we chose to use a dedicated meta-language, Skel, as its library [16] makes

defining interpreters for Skel convenient and one objective is to use the NecroCoq tool [15] to generate mechanised proofs that our derived abstract interpreters are correct.

7 Conclusion

In this paper, we propose a methodology for mechanically deriving correct abstract interpreters from mechanised semantics. Our approach is based on Skeletal Semantics and its meta-language Skel, used to write a semantic description of a language. It consists of two independent parts. First, we define an abstract interpreter for Skel which is target language-agnostic and is the core of all derived abstract interpreters from Skeletal Semantics. The abstract interpreter of Skel is proved correct with respect to the operational semantics of Skel. Second, for a given target language to analyse, abstractions must be defined. The abstract domains are defined by instantiating the unspecified types and providing comparisons and abstract unions of abstract values. The semantics of the language-specific parts are defined by instantiating the unspecified terms. By combining the abstract interpreter of Skel and the abstractions of the target language, we derive a working abstract interpreter specialised for the target language, obtained by meta-interpretation of Skel. We prove a theorem which states that the abstract interpreter of the target language is correct if the abstract instantiation of the unspecified terms are sound approximation of the concrete instantiation of the unspecified terms. We illustrate our method to build abstract interpreters on two examples: a value analysis for a small imperative language, and a CFA for λ-calculus (in the long version [7]).

The approach has been evaluated by an implementation of a tool [19] to generate abstract interpreters from any skeletal semantics. It was tested on While and λ-calculus and resulted in executable, sound analyses validating the feasibility of the approach.

The current abstract interpreters that we obtain have limitations to their precision. Part of this imprecision stems from the fact that we generate abstract interpreters for any language based on an abstract interpreter for the Skel meta-language skeletal semantics. An interesting feature of the approach is that some precision can be gained in a generic fashion by improving the underlying abstract interpretation of Skel. For example, our interval analysis for While does not refine the abstract values when entering a part of the program guarded by a condition. Take `If(Equal(x, 0), Skip, Assign(x, 0))`, evaluated in store where $\{x \mapsto \top\}$. Our abstract interpreter returns state $\{x \mapsto \top\}$. Indeed, the condition can be true or false thus both branches of the if construct are evaluated but each one is computed in the store $\{x \mapsto \top\}$ because the condition is not used to refine the abstract values. This issue can be addressed, e.g., by keeping a trace of the execution in order to know if we are computing a statement guarded by a condition. Dealing with this issue at the level of the meta-language analysis benefits all generated analyses.

References

[1] Martin Bodin, Philippa Gardner, Thomas Jensen & Alan Schmitt (2019): *Skeletal semantics and their interpretations*. *Proceedings of the ACM on Programming Languages* 3(POPL), pp. 1–31, doi:`10.1145/3291651`.

[2] Denis Bogdanas & Grigore Roşu (2015): *K-Java: A complete semantics of Java*. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 445–456, doi:`10.1145/2676726.2676982`.

[3] Patrick Cousot (1998): *The Calculational Design of a Generic Abstract Interpreter*. Marktoberdorf Course Notes.

[4] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In Robert M. Graham, Michael A. Harrison & Ravi Sethi, editors: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, ACM, pp. 238–252, doi:10.1145/512950.512973.

[5] Patrick Cousot & Radhia Cousot (1979): *Systematic Design of Program Analysis Frameworks*. In Alfred V. Aho, Stephen N. Zilles & Barry K. Rosen, editors: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, ACM Press, pp. 269–282, doi:10.1145/567752.567778.

[6] John Hughes (2000): *Generalising monads to arrows*. *Science of computer programming* 37(1-3), pp. 67–111, doi:10.1016/S0167-6423(99)00023-4.

[7] Thomas Jensen, Vincent Rébiscoul & Alan Schmitt (2023): *Deriving Abstract Interpreters from Skeletal Semantics (Long Version)*. Available at https://skeletons.inria.fr/cfa/express-sos-2023-long.pdf.

[8] Jacques-Henri Jourdan (2016): *Verasco: a formally verified C static analyzer*. Ph.D. thesis, Universite Paris Diderot-Paris VII.

[9] Sven Keidel, Casper Bach Poulsen & Sebastian Erdweg (2018): *Compositional soundness proofs of abstract interpreters*. *Proceedings of the ACM on Programming Languages* 2(ICFP), pp. 1–26, doi:10.1145/3235031.

[10] Adam Khayam, Louis Noizet & Alan Schmitt (2022): *A Faithful Description of ECMAScript Algorithms*. In: *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming*, PPDP '22, Association for Computing Machinery, New York, NY, USA, pp. 8:1–8:14, doi:10.1145/3551357.3551381.

[11] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Communications of the ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[12] Junghee Lim & Thomas Reps (2013): *TSL: A system for generating abstract interpreters and its application to machine-code analysis*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 35(1), pp. 1–59, doi:10.1145/2450136.2450139.

[13] Flemming Nielson (1982): *A denotational framework for data flow analysis*. *Acta Informatica* 18(3), pp. 265–287, doi:10.1007/BF00263194.

[14] Flemming Nielson (1989): *Two-level semantics and abstract interpretation*. *Theoretical Computer Science* 69(2), pp. 117–242, doi:10.1016/0304-3975(89)90091-1.

[15] Louis Noizet: *Necro Gallina Generator, https://gitlab.inria.fr/skeletons/necro-coq*. Available at https://gitlab.inria.fr/skeletons/necro-coq.

[16] Louis Noizet: *Necro Library, https://gitlab.inria.fr/skeletons/necro*. Available at https://gitlab.inria.fr/skeletons/necro.

[17] Louis Noizet & Alan Schmitt (2022): *Semantics in Skel and Necro*. In: *ICTCS 2022 - Italian Conference on Theoretical Computer Science*, *CEUR Workshop Proceedings* 3284, CEUR-WS.org, Rome, Italy, pp. 99–115.

[18] Grigore Roșu & Traian Florin Șerbănuță (2010): *An overview of the K semantic framework*. *The Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[19] Vincent Rébiscoul: *Abstract Interpreter Generator, https://gitlab.inria.fr/skeletons/abstract-interpreter-generator*. Available at https://gitlab.inria.fr/skeletons/abstract-interpreter-generator.

[20] David A Schmidt (1995): *Natural-semantics-based abstract interpretation (preliminary version)*. In: *International Static Analysis Symposium*, Springer, pp. 1–18, doi:10.1007/3-540-60360-3_28.