Building CFA for λ-calculus from Skeletal Semantics

Thomas Jensen, Vincent Rébiscoul, and Alan Schmitt

INRIA Rennes

Abstract

This paper describes a method to define a correct abstract interpretation from a formal description of the semantics of a programming language. Our approach is based on Skeletal Semantics. We extend it with a notion of program points, in order to differentiate two fragments of the program that are syntactically equivalent but appear at different locations. We introduce a methodology for deriving an abstract interpretation from a Skeletal Semantics that is correct by construction: given a program, abstract states are computed for each program points. We apply our method by defining a Control Flow Analysis for λ-calculus from its Skeletal Semantics.

1 Introduction

As the complexity of software increases, building static analyses becomes more and more important. Analyzers have improved for years and many companies use them to save time by detecting bugs [2], or prove some properties on their code [4]. In both cases, the soundness of the analysis is paramount. Methods to develop analyses from a description of a language are useful because they are systematic, even if done by hand. One of the most famous example is Abstract Interpretation [3], which explains how to define an abstract semantics from an operational semantics and how to prove that the abstract semantics is sound using a Galois connection. Other works used abstract interpretation, for example to build an abstract semantics from a big-step semantics [10].

In order to mechanize the design of sound analyses, one should start from a mechanized semantics. Several tools provide such semantics, such as $\mathbb{K}$ [7], a framework to mechanize semantics using rewriting rules. This makes the formal definition of a semantics easier, and allows to mechanically derive objects from the semantics: for example in $\mathbb{K}$, one can automatically derive an interpreter from the mechanization of a language. However, $\mathbb{K}$ is not an extensible framework and it is unclear how to derive an analysis from a mechanization in $\mathbb{K}$.

In this paper we describe the first steps towards the generation of analyses from a Skeletal Semantics. Skeletal Semantics [1] is a recent proposal for machine-representable semantics of programming languages, using a minimalist functional language, Skel. From the Skeletal Semantics of a language, a semantics can be derived by meta-interpretation of Skel, as a big-step semantics [6]. Abstract Interpretation is a powerful framework to build analyses, thus our work focuses on building an abstract interpretation from a Skeletal Semantics. Because both big-step semantics and abstract interpretations stem from the same syntactic object, the proof of soundness of the abstract interpretation is in large part independent from the language considered. Proving the correctness of the abstract interpretation and the big-step semantics given a particular Skeletal Semantics is therefore dependent on small lemmas only.

Our goal is to provide a methodology to easily define several semantics for a language that can be related to one another. Skeletal Semantics [1] is a recent proposal for machine-representable semantics of programming languages The Skeletal Semantics of a language is a partial description of the language. Several meanings, called interpretations, can be given to this

description like a big-step semantics [6]. However, because the description is partial, some parts are left *unspecified*, and to fully define a semantics, one needs to provide *specifications* to the unspecified parts. There are two benefits to this approach. First, the different interpretations are language independent: an interpretation can be used with any Skeletal Semantics. Second, two interpretations can be related to one another: for example, in this paper we present an abstract interpretation that is a correct approximation of the big-step interpretation.

One benefit to this approach is that the interpretations of Skel can be used for any language with a Skeletal Semantics, the definition of a semantics is done by only specifying the language dependent unspecified parts. Because it is often easy to define relations between interpretations, the different semantics of one language can also be related to by proving small lemmas on the specifications that depend on the interpretations. For example, we present a method in this paper to define an abstract interpretation from a Skeletal Semantics that is a sound approximation of the big-step semantics, provided that some lemmas about the

Contributions We propose a new interpretation of Skel that includes program points in a systematic way. We define an abstract interpretation for Skel which is correct: the set of all the executions of the big-step semantics is safely over-approximated. We provide a modular methodology to prove correctness at the skeletal meta-level. We give an example of how to define a CFA analysis for λ-calculus using the abstract interpretation of Skel. This work has been implemented in a small OCaml program which takes as inputs a Skeletal Semantics and the definitions of unspecified types and terms, and returns an abstract interpreter.

In Section 2, we introduce the syntax of Skeletal Semantics. In Section 3, we present a big-step semantics (or concrete interpretation) of Skel. In Section 4, we show how to modify the big-step semantics to use program points of a given program. In Section 5, we present an abstract interpretation of Skel, used to define abstract interpretation for any language with a Skeletal Semantics, with a theorem of soundness between the Skel abstract interpretation and big-step semantics. Finally, in Section 6, we use the abstract interpretation to define a CFA analysis for the λ-calculus

2 Skeletal Semantics and their Syntax

Skeletal Semantics is a recent approach to mechanize semantics for programming languages [1]. It uses a minimalist, functional, and strongly typed language called Skel [6]. The Necro library [5] is a tool to manipulate Skeletal Semantics. Given Skel code, it can generate Ocaml code, Coq code, step by step debuggers, and more.

The mechanization of a semantics of a language using Skeletal Semantics is done by meta-interpretation of the Skel language, therefore *interpretations* must be provided. In this paper, we will present two interpretations: a big-step semantics and an abstract interpretation.

A Skeletal Semantics is a syntactic object, written in Skel, describing a language. We present the Skeletal Semantics of the call-by-value λ-calculus with environments. We start by defining useful types for our language:

```
1 type ident
2 type env
3
4 type clos =
5 | Clos (ident, lterm, env)
6
```

```
type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)
```

It starts with two definitions of *unspecified* types, and two definitions of *specified* types. The types refer, in that order, to identifiers, environments, closures, and λ-terms. Having unspecified types is a unique trait of Skel. Keeping some types unspecified is useful because their instantiation can depend on the interpretation of Skel. Specified types are algebraic data types (ADT). The `clos` type contains only one constructor which parameter is a triplet with an identifier (representing a variable), a λ-term, and an environment, which is a usual definition for a closure. The type `lterm` is the definition of λ-terms: it can be a λ-abstraction, a variable or an application.

Moreover, a Skeletal Semantics contains terms:

```
val extEnv : (env, ident, clos) → env
val getEnv : (ident, env) → clos
```

The term $getEnv(x, e)$ is a lookup function: it returns the closure associated to x in e. The term $extEnv(e, x, v)$ returns a new environment equals to e but with the new binding where x maps to v. Terms must be explicitly typed. Moreover, both terms are *unspecified*: like types, terms do not need to be completely defined in the Skeletal Semantics. In fact, as env is unspecified, functions to access or extend it must be unspecified.

A Skeletal Semantics also contains specified terms:

```
val eval (s:env) (l:lterm): clos =
        branch
                let Lam (x, t) = l in
                Clos (x, t, s)
        or
                let Var x = l in
                getEnv (x, s)
        or
                let App (t1, t2) = l in
                let Clos (x, t, s') = eval s t1 in
                let w = eval s t2 in
                let s'' = extEnv (s', x, w) in
                eval s'' t
        end

```

This specified term is the function detailing how a `lterm` should be computed given an environment. For example, the eval function starts with a branching with three branches, and each branch is guarded by a let-binding doing pattern-matching: this particular branching acts as a case analysis. Indeed, a branching is used to cover several cases: here, there is one case per constructor in the `lterm` type. The first branch evaluates a λ-abstraction by returning a closure, the second branch evaluates a variable by using the getEnv function. The final branch evaluates an application: first t1 is evaluated, which gives a closure $Clos(x, t, s')$ representing the function with parameter x and body t, and where environment s' gives meaning to free variables in t. Then t2 is evaluated to another closure w, and s' is extended such that x maps

$$
\begin{array}{rrcl}
\textsc{Term} & t & ::= & x \mid C\ t \mid (t, \ldots, t) \mid \lambda p : \tau \to S \\
\textsc{Skeleton} & S & ::= & t_0\ t_1 \ldots t_n \mid \textbf{let}\ p\ =\ S\ \textbf{in}\ S \mid \text{branch}\ S\ \text{or}\ ..\ \text{or}\ S\ \text{end} \mid t \\
\textsc{Pattern} & p & ::= & x \mid \_ \mid C\ p \mid (p, \ldots, p) \\
\textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau, \ldots, \tau) \\
\textsc{Term decl} & r_t & ::= & \textbf{val}\ x : \tau \mid \textbf{val}\ x : \tau = t \\
\textsc{Type decl} & r_\tau & ::= & \textbf{type}\ b \\
 & & \mid & \textbf{type}\ b = |C1\ \tau_1 \ldots |Cn\ \tau_n
\end{array}
$$

Figure 1: The Syntax of Skeletal Semantics

to w, giving a new environment s". Finally, the body of the function t is evaluated with the new environment s". This should not be too surprising for people familiar with the semantics of λ-calculus with environments.

Formally, the syntax of Skel is given in Figure 1: it is similar to λ-calculus where the syntax ensures that programs are in A-Normal Form [9]. A specified *term* is either a variable, a constructor applied to a term, a tuple, or a function. Intuitively, a term is a construct that is fully computed. A *skeleton* is either an application, a let-binding, a branching, or a term. Intuitively, a skeleton is a computation. Branching is the most exotic construct of the language, it will be explained in more depth later on. Skel uses *patterns*, notably to perform pattern-matching with let-bindings. A *term declaration* introduces a top-level term. It can either be *unspecified*, in which case only its name and type are given, or *specified*, in which case the specification is a term. A *type declaration* introduces a type that may be unspecified, or it can be the declaration of an algebraic datatype.

A Skeletal Semantics is a set of type definitions and term definitions, that can be specified or unspecified. In the following, for any constructor C of a specified algebraic data type τ, we write $C : (\tau_i, \tau)$ to state that C belongs to type τ and expects an argument of type τ_i.

3 Big-step Semantics of Skel

To give meaning to a Skeletal Semantics, we provide *interpretations* of Skel. In this section, we give the big-step semantics of Skel.

The predicate $\text{na}(\tau)$ is true when τ is not an arrow type. The *arity* of a function f, $\text{arity}(f)$, is n if f has type $\tau_1 \to \ldots \to \tau_n \to \tau$, where $\text{na}(\tau)$. Let $\mathcal{S}$ be an arbitrary Skeletal Semantics. We write $\text{Funs}(\mathcal{S})$ 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 $\mathcal{S}$. The typing environment Γ gives types to the free variables of $\lambda p : \tau_1 \to S_0$. Typing rules are given in the Appendix B and $\text{Funs}(\cdot)$ is formally defined in the Appendix C.

3.1 From Types to Concrete Values

We call the interpretation of types the function $V(\tau)$, that given a type τ returns the set of values of that type. These sets are defined with the relation $\vdash \cdot \in V(\cdot)$ in Figure 2. These rules are language independent and are completed with language-dependent rules upon instantiation.

$$\frac{\forall 1 \leq i \leq n \qquad \vdash v_i \in V(\tau_i)}{\vdash (v_1, \ldots, v_n) \in V(\tau_1 \times \cdots \times \tau_n)}\ \textsc{Tuple} \qquad \frac{\vdash v \in V(\tau) \qquad C : (\tau, \tau_a)}{\vdash C\ v \in V(\tau_a)}\ \text{ADT}$$

$$\frac{(\Gamma, \lambda p : \tau_1 \to S) \in \text{Funs}(\mathcal{S}) \qquad \Gamma \vdash E \qquad \Gamma + p \leftarrow \tau_1 \vdash S : \tau_2}{\vdash (p, S, E) \in V(\tau_1 \to \tau_2)}\ \textsc{Clos}$$

$$\frac{\textbf{val}\ f : \tau_1 \to \cdots \to \tau_n \to \tau\ [= t] \in \mathcal{S} \qquad \text{na}(\tau)}{\vdash (f, \text{arity}(f)) \in V(\tau_1 \to \cdots \to \tau_n \to \tau)}\ \textsc{Def}$$

$$\frac{\text{dom}\ \Gamma = \text{dom}\ E \qquad \forall x \in \text{dom}\ E, \Gamma \vdash E(x) : \Gamma(x)}{\Gamma \vdash E}\ \textsc{IEnv}$$

Figure 2: Rules to Build Values

The Tuple and ADT rules build tuples and values of ADTs. The Clos rule builds a closure, which is a a triplet (p, S, E). p is the pattern that contains the parameters of the function, S is the body of the function and E an environment for the free variables in S. Note that this environment must be compatible with the typing of the free variables of the function (Rule IEnv). The Def rule builds named closures for application of unspecified and specified terms with arrow type. The closure contains only the name and the arity of the function.

An instantiation of a Skeletal Semantics must define the values belonging to unspecified types. We now show how this is done by example. In the case of λ-calculus, the unspecified types are `ident` and `env`, thus we provide rules to build these values.

$$\frac{id \in \mathcal{V} = \{\, x, y, z ... \,\}}{\vdash id \in V(\texttt{ident})}\ \textsc{Ident} \qquad \frac{}{\vdash [\,] \in V(\texttt{env})}\ \textsc{Env-Empty}$$

$$\frac{\vdash id \in V(\texttt{ident}) \qquad \vdash c \in V(\texttt{clos}) \qquad \vdash e \in V(\texttt{env})}{\vdash (id, c) :: e \in V(\texttt{env})}\ \textsc{Env-Cons}$$

Identifiers are variables from a countable set of variables $\mathcal{V}$, and an environment is an association list: it can be empty or built from a new binding and another environment.

3.2 Interpretation of Unspecified Terms

If X is a set, $\mathcal{P}_f(X)$ is the set of finite parts of X.

Now that values are defined, there remains to give definitions to unspecified terms. Take an unspecified term $\textbf{val}\ t : \tau_1 \to \cdots \to \tau_n \to \tau$ such that $\text{na}(\tau)$, then an instantiation of t, written $[\![t]\!]$, is a function such that: $[\![t]\!] \in (V(\tau_1) \times \cdots \times V(\tau_n)) \to \mathcal{P}_f(V(\tau))$. In particular, if $\textbf{val}\ t : \tau$ and $\text{na}(\tau)$, then $[\![t]\!] \subseteq V(\tau)$. Allowing the specification of a term to be a function returning a set is useful to model non-determinism.

For our λ-calculus, the specifications of unspecified terms are:

$$
\begin{aligned}
&[\![getEnv]\!](x, (x, c) :: e) = \{\, c \,\} \\
&[\![getEnv]\!](x, (y, c) :: e) = [\![getEnv]\!](x, e) \qquad\qquad x \neq y \\
&[\![getEnv]\!](x, [\,]) = \{\,\}
\end{aligned}
$$

$$
[\![extEnv]\!](e, x, c) = (x, c) :: e
$$

The interpretation $[\![getEnv]\!]$(x,e) is defined with three equations: the first equation returns the closure when the correct binding is at the head of the environment, the second equation is a recursive call when the first binding is not the correct one, and in last equation, getEnv returns the empty set because the environment is empty. $[\![extEnv]\!]$(e, x, c) returns a similar environment to e, but with a new binding where x is associated to c.

3.3 Big-step Semantics

We now define 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 in Figure 3a.

There are four rules to evaluate variables, depending on how the variable was defined and its type. If the variable was defined in a 'let' expression, the environment E maps the variable to its value (Rule Var). Otherwise, assuming the Skeletal Semantics is well typed, the variable must be the name of a specified or unspecified term. If this term has an arrow type, we simply return a named closure, whether the variable is specified or not. A named closure is a pair of the name and arity of the function (Rule TermClos). If the variable is declared in the Skeletal Semantics, then it is bound to a closed term t, which is evaluated in the empty environment because it only depends on term declarations of the Skeletal Semantics (Rule TermSpec). If the variable is unspecified, a value of the provided instantiation is returned (Rule TermUnspec). Rules Const and Tuple are usual. Rule Clos returns a skeletal closure. One may observe a similarity between this meta-rule and the corresponding object rule in the λ-calculus; this is because Skel is a meta-language that is an extension of the λ-calculus, but the meta-language and the object language should not be confused.

We now turn to the evaluation of skeletons. Rule LetIn is usual. Rule Branch evaluates a branching by non-deterministically returning the result of the evaluation of a branch. To evaluate an application, we define another relation, defined in Figure 3b with four rules. Rule Base returns the result when all arguments have been processed. Rule Clos is the application of a closure. Rules Spec and Unspec are the application of a named closure to a list of arguments. The first evaluates the corresponding definition while the second one uses the user-provided instantiation. Non-specified functions may be non-deterministic since their instantiation may return a set. Note that we only consider full application of named functions: all arguments must be provided. Extending this semantics to partial application is easy, although verbose, but it is not necessary to describe our contributions. Finally, the pattern matching rules for environment extension are given in Figure 4.

$$\frac{E(x) = v}{E, x \Downarrow v}\ \textsc{Var} \qquad \frac{\textbf{val}\ f : \tau_1 \to \cdots \to \tau_n \to \tau [= t] \qquad \mathrm{na}(\tau) \qquad n \geq 1}{E, f \Downarrow (f, n)}\ \textsc{TermClos}$$

$$\frac{\textbf{val}\ x : \tau = t \in \mathcal{S} \qquad \mathrm{na}(\tau) \qquad \emptyset, t \Downarrow v}{E, x \Downarrow v}\ \textsc{TermSpec}$$

$$\frac{\textbf{val}\ x : \tau \in \mathcal{S} \qquad \mathrm{na}(\tau) \qquad v \in [\![x]\!]}{E, x \Downarrow v}\ \textsc{TermUnspec} \qquad \frac{E, t \Downarrow v}{E, (\mathrm{C}\, t) \Downarrow \mathrm{C}\, v}\ \textsc{Const}$$

$$\frac{E, t_1 \Downarrow v_1 \qquad \ldots \qquad E, t_n \Downarrow v_n}{E, (t_1, \ldots, t_n) \Downarrow (v_1, \ldots, v_n)}\ \textsc{Tuple} \qquad \frac{}{E, (\lambda p : \tau \to S) \Downarrow (p, S, E)}\ \textsc{Clos}$$

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

$$\frac{\forall i \in [0..n].\ E, t_i \Downarrow v_i \qquad v_0\ v_1\ \ldots\ v_n \Downarrow_{app} w}{E, (t_0\ t_1 \ldots t_n) \Downarrow w}\ \textsc{App}$$

(a) Rules for the Big-step Semantics of Skeletons and Terms

$$\frac{}{v \Downarrow_{app} v}\ \textsc{Base} \qquad \frac{\vdash E + p \leftarrow v_1 \rightsquigarrow E' \qquad E', S \Downarrow v \qquad v\ v_2 \ldots v_n \Downarrow_{app} w}{(p, S, E)\ v_1 \ldots v_n \Downarrow_{app} w}\ \textsc{Clos}$$

$$\frac{\textbf{val}\ f : \tau_1 \to \cdots \to \tau_n \to \tau = t \in \mathcal{S} \qquad \mathrm{na}(\tau) \qquad \emptyset, t \Downarrow v \qquad v\ v_1 \ldots v_n \Downarrow_{app} w}{(f, n)\ v_1 \ldots v_n \Downarrow_{app} w}\ \textsc{Spec}$$

$$\frac{\textbf{val}\ f : \tau_1 \to \cdots \to \tau_n \to \tau \in \mathcal{S} \qquad \mathrm{na}(\tau) \qquad w \in [\![f]\!](v_1, \ldots, v_n)}{(f, n)\ v_1 \ldots v_n \Downarrow_{app} w}\ \textsc{Unspec}$$

(b) Rules for Application for the Big-step Semantics

Figure 3: Interpretation rules of the Big-step Semantics

$$\frac{}{\vdash E + \_ \leftarrow v \rightsquigarrow E}\ \textsc{asn-wildcard} \qquad \frac{}{\vdash E + x \leftarrow v \rightsquigarrow (x, v) :: E}\ \textsc{asn-var}$$

$$\frac{\vdash E + p \leftarrow v \rightsquigarrow E'}{\vdash E + C\, p \leftarrow C\, v \rightsquigarrow E'}\ \textsc{asn-constr}$$

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

Figure 4: Rule of Extension of Environment through Pattern Matching

4 Big-step Semantics with Program Points

4.1 Program Points and New Values

We present our first contribution, which is a method to use *program points* in the big-step semantics of a Skeletal Semantics. This methodology is reused later for the abstract interpretation of Skel. Let $\mathcal{S}$ be a Skeletal Semantics. It usually contains one or more specified algebraic data types that define the syntax of the language, i.e., *programs*. We call these types *program types*. In the case of λ-calculus, `lterm` is the only program type. Let τ a program type in $\mathcal{S}$ and $\mathbf{prg} \in V(\tau)$ a program, we can refer to sub-terms of **prg** using program points. A program point is a path from the root of **prg** to one of its sub-term. Given a program point pp, $\mathbf{prg}@\,\mathrm{pp}$ is the sub-term of **prg** obtained by following path pp. A program point pp is an element of $\mathtt{ppt} = \mathbb{N}^*$, the set program points. ϵ is the empty path and $i \cdot \mathrm{pp}$ is a path with first element i, and pp is the rest of the path. The @ operator is formally defined as:

$$
\begin{aligned}
&v@\epsilon = v \\
&C(v_0, \ldots, v_{n-1})@i \cdot \mathrm{pp} = v_i@\,\mathrm{pp} \qquad\qquad \text{when } 0 \leq i \leq n-1
\end{aligned}
$$

To give an example, for λ-calculus, let $\mathbf{prg} = Lam(x, Var\ x)$. Then, $\mathbf{prg}@0 = x$ and $\mathbf{prg}@1 = Var\ x$.

Our approach is to replace values of program types with program points. Each program point refers to a sub-term of the main program: **prg**, which is now a parameter of the interpretation. Let $\mathcal{T}$ be the set of program types from $\mathcal{S}$ and **prg** a program. Therefore, all values of $\tau \in \mathcal{T}$ are program points and represent sub-terms of **prg** . In particular, because **prg** is a program, its type should be in $\mathcal{T}$.

For each type τ, a set of values is built using previously defined rules 2 except for program types that are defined by the equation:

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

The set $V^{\mathtt{ppt}}_{\mathbf{prg}}(\tau)$ is the set of program points that refer to sub-terms of **prg** of type τ. This is an important restriction because if $\tau \in \mathcal{T}$, then a value $v \in V^{\mathtt{ppt}}_{\mathbf{prg}}(\tau)$ is a program point and therefore necessarily represent a fragment of **prg** and not an arbitrary program.

$\mathcal{T}$ is a set because some languages may have many types for different parts of programs: an imperative language would need at least one type for statements and one type for expressions. Let us give an example with the λ-calculus. We take our program to be $\mathbf{prg} = App(Lam(x, Var\ x), Var\ y)$ and $\mathcal{T} = \{\, lterm \,\}$. The interpretation of the λ-terms is defined in the next equation. The program points are underlined to differentiate them from natural numbers.

$$
V^{\mathtt{ppt}}_{\mathbf{prg}}(lterm) = \{\, \underline{\epsilon}, \underline{0}, \underline{1}, \underline{01} \,\}
$$

For example, we have $\mathbf{prg}@\underline{1} = Var\ y$

The interpretation of unspecified types does not change and other types are built using previous rules in Figure 2. However, we assume new specifications of unspecified terms. Indeed, some unspecified terms may depend on types in $\mathcal{T}$, therefore, for all unspecified terms x in $\mathcal{S}$, we suppose that new specifications are provided: $[\![x]\!]^{\mathtt{ppt}}$.

4.2 Pattern Matching with Program Points

The interpretation of skeletons with program points does not fundamentally change: only the pattern matching is modified and uses an unfolding mechanism. Unfolding matches a program

$$\mathrm{NC}(\tau_1 \to \tau_2) = \left\{ (f, n) \;\middle|\; \begin{array}{l} \textbf{val}\ f : \tau_1 \to \tau_2[= t] \in \mathcal{S} \\ \mathrm{arity}(f) = n \end{array} \right\}$$

$$\mathrm{C}(\tau_1 \to \tau_2) = \left\{ (p, S, E^\sharp) \;\middle|\; \begin{array}{l} \exists (\Gamma, \lambda p : \tau_1 \to S) \in \mathrm{Funs}(\mathcal{S}), \\ \Gamma \vdash E^\sharp \wedge \Gamma + p \leftarrow \tau_1 \vdash S : \tau_2 \end{array} \right\}$$

Figure 5: Sets of Abstract Named Closures and Closures

point with a constructor pattern and is described by the following rule.

$$\frac{\begin{array}{c} \textbf{prg}@\,\mathrm{pp} = C\,(v'_0, \ldots, v'_{n-1}) \qquad C : (\tau_0 \times \cdots \times \tau_{n-1}, \tau) \\ v_j = \textbf{if}\ \tau_j \in \mathcal{T}\ \textbf{then}\ \mathrm{pp} \cdot j\ \textbf{else}\ v'_j \qquad \mathcal{T}, \textbf{prg} \vdash E + p \leftarrow (v_0, \ldots, v_{n-1}) \rightsquigarrow E' \end{array}}{\mathcal{T}, \textbf{prg} \vdash E + C\,p \leftarrow \mathrm{pp} \rightsquigarrow E'}\ \textsc{asn-unfold}$$

When a program point pp is matched with a constructor pattern $C\ p$, $\textbf{prg}@\,\mathrm{pp}$ is expected to have the form $C(v'_0, \ldots, v'_{n-1})$. Then, for $0 \leq j \leq n-1$, v_j is $\mathrm{pp} \cdot j$ if v'_j has a program type $\tau_j \in \mathcal{T}$, or v'_j otherwise. Then the pattern p is matched with $(v_1, \ldots, v_n)$. The unfolding rule exhibits the constructor and its parameters at program point pp, then continues the pattern matching recursively.

We give an example for λ-calculus, let $\textbf{prg} = App(Lam(x, Var\ x), Var\ y)$. To compute $\mathcal{T}, \textbf{prg} \vdash E + Lam(p) \leftarrow \underline{0} \rightsquigarrow E'$ where p is a pattern, by applying the ASN-UNFOLD rule, it comes that: $\mathcal{T}, \textbf{prg} \vdash E + p \leftarrow (x, \underline{01}) \rightsquigarrow E'$ Indeed, $\textbf{prg}@0 = Lam(x, Var\ x)$ and $Var\ x$ has program point 01, therefore p is matched with $(x, \underline{01})$

The big-step semantics of skeletons and the big-step semantics of skeletons with program points are closely related. We formalize this connection through a theorem presented in Appendix D.

5 Abstract Interpretation

We define an abstract interpretation of Skel that is sound with respect to the big-step semantics of Section 4. The abstract interpretation is used in the next section to define a Control Flow Analysis for λ-calculus.

We define the abstract values, comparison functions and upper bounds in Section 5.1. We define the state of the abstract interpretation in Section 5.2. We present the interpretation rules of skeletons for an abstract interpretation in Section 5.3, We present how the abstract values are linked to the concrete values by defining concretization functions in Section 5.4. Finally, we formulate a theorem of soundness in Section 5.5. In the following sections, $\mathcal{S}$ denotes an arbitrary Skeletal Semantics.

5.1 Definition of Abstract Values

The abstract values are defined similarly to the concrete ones. In Figure 5 we define the set of abstract named closures and the set of abstract closures. Abstract named closures are pairs of a name of a function defined in the skeletal semantics and its arity. Abstract closures are a triplet with a pattern, a skeleton and an abstract environment. The abstract environment is defined later in the this section.

$$\frac{}{\vdash^\sharp \bot_\tau \in V^\sharp(\tau)}\ \textsc{Bottom} \qquad \frac{}{\vdash^\sharp \top_\tau \in V^\sharp(\tau)}\ \textsc{Top}$$

$$\frac{\forall(v_1^\sharp,\ldots,v_n^\sharp) \in t^\sharp \qquad \forall 1 \leq i \leq n,\ \vdash^\sharp v_i^\sharp \in V^\sharp(\tau_i) \wedge v_i^\sharp \neq \bot_{\tau_i}}{\vdash^\sharp t^\sharp \in V^\sharp(\tau_1 \times \cdots \times \tau_n)}\ \textsc{Tuple}$$

$$\frac{\vdash^\sharp v^\sharp \in V^\sharp(\tau) \qquad C : (\tau, \tau_a) \qquad v^\sharp \neq \bot_\tau}{\vdash^\sharp C\ v^\sharp \in V^\sharp(\tau_a)}\ \text{ADT}$$

$$\frac{nc \subseteq \text{NC}(\tau_1 \to \tau_2) \qquad c \subseteq \text{C}(\tau_1 \to \tau_2)}{\vdash^\sharp c \cup nc \in V^\sharp(\tau_1 \to \tau_2)}\ \textsc{Funs}$$

$$\frac{\text{dom}\ \Gamma = \text{dom}\ E^\sharp \qquad \forall x \in \text{dom}\ E^\sharp,\ \Gamma \vdash E^\sharp(x) : \Gamma(x)}{\Gamma \vdash E^\sharp}\ \textsc{IEnv}$$

Figure 6: Rules to Build Abstract Values

The rules to define abstract values are presented on Figure 6. The Bottom and Top rules define a least and a greatest element for each type. The Tuple rule define the abstract tuples, that are a set of tuples of abstract values. Because one of our target is CFA analysis for λ-calculus, having a set is necessary not to loose too much precision during the analysis. On line 24 of the Skeletal Semantics of the λ-calculus A, *eval s t*1 returns a closure, $Clos(x, t, s)$ which is essentially a triplet. The relation between identifier, term and environment must be preserved. Values of algebraic types, defined by the rule ADT, are constructors applied to abstract values. Abstract functions, defined by the rule Funs, is the union of a subset of abstract named closures, and a subset of abstract closures. For instance for the λ-calculus, the set $\{(\texttt{eval}, 2), (\texttt{s}, \lambda \texttt{l:lterm}.S_{eval}, \emptyset)\}$ belongs to $V^\sharp(\texttt{env} \to \texttt{lterm} \to \texttt{clos})$, where S_{eval} is the body of the `eval` specified term. Finally, the rule IEnv defines abstract environments, that are partial functions mapping skeletal variables to abstract values.

To compare abstract values, we define partial orders. For every unspecified type τ_u, we assume comparison function $\sqsubseteq^\sharp_{\tau_u}$ which is an order and with the constraint that $\top_{\tau_u}$ and $\bot_{\tau_u}$ are the greatest and smallest elements of $V^\sharp_{\textbf{prg}}(\tau_u)$ respectively. For every other types, the comparison function is the smallest order that satisfies the following equations:

$$\begin{aligned}
C\ v^\sharp \sqsubseteq^\sharp_{\tau_a} C\ w^\sharp &\iff v^\sharp \sqsubseteq^\sharp_\tau w^\sharp \text{ with } C : (\tau, \tau_a) \\
t_1^\sharp \sqsubseteq^\sharp_{\tau_1 \times \cdots \times \tau_n} t_2^\sharp &\iff \forall(v_1^\sharp,\ldots,v_n^\sharp) \in t_1^\sharp, \exists(w_1^\sharp,\ldots,w_n^\sharp) \in t_2^\sharp, \forall i, 1 \leq i \leq n,\ v_i^\sharp \sqsubseteq^\sharp_{\tau_i} w_i^\sharp \\
F_1 \sqsubseteq^\sharp_{\tau_1 \to \tau_2} F_2 &\iff \begin{cases} (f,n) \in F_1 \implies (f,n) \in F_2 \\ (p, S, E_1^\sharp) \in F_1 \implies \exists(p, S, E_2^\sharp) \in F_2,\ E_1^\sharp \sqsubseteq^\sharp_{env} E_2^\sharp \end{cases} \\
E_1^\sharp \sqsubseteq^\sharp_{env} E_2^\sharp &\iff \Gamma \vdash E_1^\sharp \wedge \Gamma \vdash E_2^\sharp \wedge \forall x \in \text{dom}\ E_1^\sharp,\ E_1^\sharp(x) \sqsubseteq^\sharp_{\Gamma(x)} E_2^\sharp(x) \\
& \quad v^\sharp \sqsubseteq^\sharp_\tau \top_\tau \\
& \quad \bot_\tau \sqsubseteq^\sharp_\tau v^\sharp
\end{aligned}$$

To compare algebraic values, their parameters are compared. Tuples are compared by checking

that all tuples of abstract values of the left tuple are smaller than a tuple of abstract values in the right tuple. 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 each type, top and bottom are respectively the smallest and greatest elements of the type.

For each type, an upper bound (or join) is defined. For every non-specified type τ_u, we assume an upper bound $\sqcup^\sharp{}_{\tau_u}$. We define an upper bound $\sqcup^\sharp{}_\tau$ for every other types.

$$
\begin{aligned}
(C\ v^\sharp)\sqcup^\sharp{}_{\tau_a}(C\ w^\sharp) &= C\ (v^\sharp \sqcup^\sharp{}_\tau w^\sharp) \text{ with } C : (\tau, \tau_a) \\
(C\ v^\sharp)\sqcup^\sharp{}_{\tau_a}(D\ w^\sharp) &= \top_{\tau_a} \text{ with } C : (\tau, \tau_a) \wedge D : (\tau', \tau_a) \\
t_1^\sharp \sqcup^\sharp{}_{\tau_1 \times \cdots \times \tau_n} t_2^\sharp &= t_1^\sharp \cup t_2^\sharp \\
F_1 \sqcup^\sharp{}_{\tau_1 \to \tau_2} F_2 &= F_1 \cup F_2 \\
E_1^\sharp \sqcup^\sharp{}_{env} E_2^\sharp &= \left\{ x \in \text{dom } E_1^\sharp \mapsto E_1^\sharp(x) \sqcup^\sharp E_2^\sharp(x) \right\} \\
v^\sharp \sqcup^\sharp{}_\tau \top_\tau = \top_\tau \sqcup^\sharp{}_\tau v^\sharp &= \top_\tau \\
v^\sharp \sqcup^\sharp{}_\tau \bot_\tau = \bot_\tau \sqcup^\sharp{}_\tau v^\sharp &= v^\sharp
\end{aligned}
$$

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

5.2 State of the Abstract Interpretation

The state of the abstract interpretation $\mathcal{A}$ is a machine representable state that contains information collected throughout the abstract interpretation. It is dependent on the analysis and the language, and therefore is non-generic and must be specified. To give an example, when defining a CFA in the next section, the abstract state will contain a mapping from program points to sets of λ-abstraction, that can be viewed as the potential results when evaluating a sub-term at some program point of the analyzed program. We require an order on the abstract states. Intuitively, we should only add information in the states throughout the analysis, and therefore at each step of the analysis, the states should only increase.

A state contains several components: $\mathcal{A} \cdot c$ is the component c of the abstract state $\mathcal{A}$. The notation $\{\ \mathcal{A}\ \textbf{with}\ c = v\ \}$ denotes a new state equals to $\mathcal{A}$, excepts for the c component which is equal to v.

5.3 The Abstract Interpretation of Skel

The abstract interpretation maintains a callstack of specified function. This is used for loop detection and prevent infinite computations: the idea is to inspect the callstack at each call to a specified function to detect identical nested calls. Callstacks are ordered list of frames,

formally defined as:

$$\frac{}{\epsilon \in \textbf{callstack}}$$

$$\frac{\mathcal{A} \text{ an abstract state} \quad \textbf{val} f : \tau_1 \to \cdots \to \tau_n \to \tau = t \in \mathcal{S} \qquad \text{na}(\tau) \qquad v_i \in V^\sharp_{\textbf{prg}}(\tau_i) \qquad \pi \in \textbf{callstack}}{(f, \mathcal{A}, [v_1, \ldots, v_n]) :: \pi \in \textbf{callstack}}$$

The abstract interpretation of skeletons is given on Figure 7a. 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. The pattern matching now returns sets of environments rather than one, and this will be detailed later, but in consequence the LETIN rule may evaluate S_2 in several abstract environments. The APP rule evaluates all terms and pass a list of values to the application relation. There are separate rules to handle applications on Figure 7b. 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 processed. Because the extension of environments returns a set of abstract environment, the CLOS rule is modified accordingly and the body of the function S is evaluated in all abstract environments. The SPEC rule evaluates the call to a specified function by doing three things:

- it uses an $\text{update}^{in}_f(\cdot)$ function which is language dependent and must be specified. It can modify the arguments and the state of the abstract interpretation.
- the call is performed (a frame is added to the callstack at that point).
- an $\text{update}^{out}_f(\cdot)$ function can modify the state of the interpretation and the returned value.

These update functions are used to maintain invariants and update the state of the abstract interpretation. An example of their use will be presented in the next section. The update functions must respect the following constraints to ensure soundness:

Remark 1.

$$\text{update}^{in}_f(\mathcal{A}, [v^\sharp_1, \ldots, v^\sharp_k]) = [v'^\sharp_1, \ldots, v'^\sharp_k], \mathcal{A}' \implies (v^\sharp_1, \ldots, v^\sharp_k) \sqsubseteq^\sharp (v'^\sharp_1, \ldots, v'^\sharp_k) \wedge \mathcal{A} \sqsubseteq^\sharp \mathcal{A}'$$
$$\text{update}^{out}_f(\mathcal{A}, [v^\sharp_1, \ldots, v^\sharp_k], v^\sharp) = v'^\sharp, \mathcal{A}' \implies v^\sharp \sqsubseteq^\sharp v'^\sharp \wedge \mathcal{A} \sqsubseteq^\sharp \mathcal{A}'$$

The extension of environments, or pattern matching, presented on Figure 8 is modified such that it returns a set of abstract environments. This is necessary because our abstraction of tuples is a finite set of tuples of abstract values. In order not to lose too much precision, we return one abstract environment per tuple of abstract values in our abstract tuple. The rules are similar to the big-step semantics, excepts that there are two rules for tuples: the rule ASN-TUPLE-SINGLETON extends the environment with a tuple of abstract values, the ASN-TUPLE forwards all tuples of abstract values in $t^\sharp$ to the ASN-TUPLE-SINGLETON rule.

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

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

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

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

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

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

(a) Rules for the Abstract Interpretation of Skeletons and Terms

$$\frac{v_0^\sharp = \bigcup_{i=1}^{n} \{\, w_i \,\} \qquad \pi, \mathcal{A}, w_i\ v_1^\sharp \dots v_n^\sharp \Downarrow_{app} v_{w_i}^\sharp, \mathcal{A}_i}{\pi, \mathcal{A}, v_0^\sharp\ v_1^\sharp \dots v_n^\sharp \Downarrow_{app} \sqcup^\sharp v_{w_i}^\sharp, \sqcup^\sharp \mathcal{A}_i}\ \textsc{App-Set} \qquad \frac{}{\pi, \mathcal{A}, v^\sharp \Downarrow_{app} v^\sharp, \mathcal{A}}\ \textsc{Base}$$

$$\frac{\begin{array}{c}\mathcal{T}, \mathbf{prg} \vdash E^\sharp + p \leftarrow^\sharp v_1^\sharp \rightsquigarrow \left\{\, E_1^\sharp, \dots, E_m^\sharp \,\right\} \\ \forall E_i^\sharp \in \left\{\, E_1^\sharp, \dots, E_m^\sharp \,\right\} \qquad \pi, \mathcal{A}, E_i^\sharp, S \Downarrow w_i^\sharp, \mathcal{A}_i \qquad \pi, \mathcal{A}_i, w_i^\sharp\ v_2^\sharp \dots v_n^\sharp \Downarrow_{app} u_i^\sharp, \mathcal{A}_i'\end{array}}{\pi, \mathcal{A}, (p, S, E^\sharp)\ v_1^\sharp \dots v_n^\sharp \Downarrow_{app} \sqcup^\sharp u_i^\sharp, \sqcup^\sharp \mathcal{A}_i'}\ \textsc{Clos}$$

$$\frac{\begin{array}{c}\mathbf{val}\ f : \tau_1 \to \cdots \to \tau_n \to \tau = t \in \mathcal{S} \\ \mathrm{na}(\tau) \qquad \emptyset, t \Downarrow v^\sharp \qquad \mathrm{update}_f^{in}(\mathcal{A}, [v_1^\sharp, \dots, v_n^\sharp]) = \mathcal{A}', [v_1'^\sharp, \dots, v_n'^\sharp] \\ (f, [v_1'^\sharp, \dots, v_n'^\sharp]) \notin \pi \qquad (f, [v_1'^\sharp, \dots, v_n'^\sharp]) :: \pi, \mathcal{A}', v^\sharp\ v_1'^\sharp \dots v_n'^\sharp \Downarrow_{app} w^\sharp, \mathcal{A}'' \\ \mathrm{update}_f^{out}(\mathcal{A}'', [v_1'^\sharp, \dots, v_n'^\sharp], w^\sharp) = w'^\sharp, \mathcal{A}'''\end{array}}{\pi, \mathcal{A}, (f, n)\ v_1^\sharp \dots v_n^\sharp \Downarrow_{app} w'^\sharp, \mathcal{A}'''}\ \textsc{Spec}$$

$$\frac{\begin{array}{c}\mathbf{val}\ f : \tau_1 \to \cdots \to \tau_n \to \tau = t \in \mathcal{S} \qquad \mathrm{na}(\tau) \qquad \emptyset, t \Downarrow v^\sharp \\ \mathrm{update}_f^{in}(\mathcal{A}, [v_1^\sharp, \dots, v_n^\sharp]) = \mathcal{A}', [v_1'^\sharp, \dots, v_n'^\sharp] \qquad (f, [v_1'^\sharp, \dots, v_n'^\sharp]) \in \pi\end{array}}{\pi, \mathcal{A}, (f, n)\ v_1^\sharp \dots v_n^\sharp \Downarrow_{app} \bot, \mathcal{A}'}\ \textsc{Spec-Loop}$$

$$\frac{\mathbf{val}\ f : \tau_1 \to \cdots \to \tau_n \to \tau \in \mathcal{S} \qquad \mathrm{na}(\tau) \qquad [\![f]\!]^\sharp(\mathcal{A}, v_1^\sharp, \dots, v_n^\sharp) = w^\sharp, \mathcal{A}'}{\pi, \mathcal{A}, (f, n)\ v_1^\sharp \dots v_n^\sharp \Downarrow_{app} w^\sharp, \mathcal{A}'}\ \textsc{Unspec}$$

(b) Rules for the application of the Abstract Interpretation

Figure 7: Abstract Interpretation of Skel

$$\frac{}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + \_ \leftarrow^\sharp v \rightsquigarrow \{ E^\sharp \}} \text{ ASN-WILDCARD}$$

$$\frac{}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + x \leftarrow^\sharp v \rightsquigarrow \{ (x, v^\sharp) :: E^\sharp \}} \text{ ASN-VAR}$$

$$\frac{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + p \leftarrow^\sharp v \rightsquigarrow \xi}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + C\, p \leftarrow^\sharp C\, v \rightsquigarrow \xi} \text{ ASN-CONSTR}$$

$$\frac{\begin{array}{c}\mathcal{T}, \mathbf{prg} \vdash E^\sharp + p_1 \leftarrow^\sharp v_1 \rightsquigarrow \xi_1 \\ \ldots \qquad \mathcal{T}, \mathbf{prg} \vdash \xi_{n-1} + p_n \leftarrow^\sharp v_n \rightsquigarrow \xi_n\end{array}}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + (p_1, \ldots, p_n) \leftarrow^\sharp (v_1, \ldots, v_n) \rightsquigarrow \xi_n} \text{ ASN-TUPLE-SINGLETON}$$

$$\frac{(v_1, \ldots, v_n) \in t \qquad \mathcal{T}, \mathbf{prg} \vdash E^\sharp + (p_1, \ldots, p_n) \leftarrow^\sharp (v_1, \ldots, v_n) \rightsquigarrow \xi_{v_1,\ldots,v_n}}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + (p_1, \ldots, p_n) \leftarrow^\sharp t \rightsquigarrow \bigcup_{(v_1,\ldots,v_n) \in t} \xi_{(v_1,\ldots,v_n)}} \text{ ASN-TUPLE}$$

$$\frac{\begin{array}{c}\mathbf{prg}@\,\mathrm{pp} = C\,(v'_1, \ldots, v'_n) \qquad C : (\tau_1 \times \cdots \times \tau_n, \tau) \\ v_j = \mathbf{if}\ \tau_j \in \mathcal{T}\ \mathbf{then}\ \mathrm{pp} \cdot j\ \mathbf{else}\ v'_j \qquad \mathcal{T}, \mathbf{prg} \vdash E^\sharp + p \leftarrow^\sharp \{ (v_1, \ldots, v_n) \} \rightsquigarrow \xi\end{array}}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + C\, p \leftarrow^\sharp \mathrm{pp} \rightsquigarrow \xi} \text{ ASN-UNFOLD}$$

Figure 8: Extension of Environment through Pattern Matching for Abstract Interpretation

5.4 Concretization Function

To define concretization functions for abstract values, we assume monotonic concretization functions for abstract values of non-specified types. Concretization functions are defined for every specified types, and take the state of the abstract interpretation as a parameter: $\gamma_\tau(\mathcal{A}, \cdot) : V^\sharp_{\mathbf{prg}}(\tau) \to \mathcal{P}_f(V^{\mathtt{ppt}}_{\mathbf{prg}}(\tau))$. We also define a function of concretization γ_{env} which maps abstract skeletal environments to concrete skeletal environments.

$$\begin{aligned}
\gamma_{\tau_a}(\mathcal{A}, C\, v^\sharp) &= \{ C\, v \mid C : (\tau, \tau_a),\ v \in \gamma_\tau(\mathcal{A}, v^\sharp) \} \\
\gamma_{\tau_1 \times \cdots \times \tau_n}(\mathcal{A}, t^\sharp) &= \bigcup_{(v_1^\sharp, \ldots, v_n^\sharp) \in t^\sharp} \gamma_{\tau_1}(\mathcal{A}, v_1^\sharp) \times \cdots \times \gamma_{\tau_n}(\mathcal{A}, v_n^\sharp) \\
\gamma_{\tau_1 \to \tau_2}(\mathcal{A}, F) &= \{ (f, n) \mid (f, n) \in F \} \cup \{ (p, S, E) \mid (p, S, E^\sharp) \in F \wedge E \in \gamma_{env}(\mathcal{A}, E^\sharp) \} \\
\gamma_{env}(\mathcal{A}, E^\sharp) &= \left\{ E \;\middle|\; \begin{array}{c} \Gamma \vdash E \quad \wedge \quad \Gamma \vdash E^\sharp \quad \wedge \quad \Gamma(x) = \tau \\ \mathrm{dom}\ E = \mathrm{dom}\ E^\sharp, E(x) \in \gamma_\tau(\mathcal{A}, E^\sharp(x)) \end{array} \right\} \\
\gamma(\mathcal{A}, \bot_\tau) &= \emptyset \\
\gamma(\mathcal{A}, \top_\tau) &= V^\sharp(\tau)
\end{aligned}$$

Lemma 1. If for all unspecified types τ_u, γ_{τ_u} is monotonic on both arguments, then for all τ, γ_τ is also monotonic on both arguments.

5.5 Correctness of the Abstract Interpretation

Definition 1. Let f be an unspecified term of type $\tau_1 \to \cdots \to \tau_n \to \tau$ where $\mathrm{na}(\tau)$. We say that $[\![f]\!]^\sharp$ is a *sound approximation* of $[\![f]\!]^{\mathtt{ppt}}$ iff $\forall v_i \in V^{\mathtt{ppt}}_{\mathbf{prg}}(\tau_i)$, $\forall v_i^\sharp \in V^\sharp_{\mathbf{prg}}(\tau_i)$, and for all abstract state $\mathcal{A}$, if

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

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

Theorem 1. Suppose $E, S \Downarrow v$ and $\epsilon, \mathcal{A}_0, E^\sharp, S \Downarrow^\sharp v^\sharp, \mathcal{A}$ and $\Gamma \vdash E$ and $\Gamma \vdash E^\sharp$ and $\Gamma \vdash S : \tau$ and $E \in \gamma(\mathcal{A}, E^\sharp)$.
Suppose $\forall (val\ x : \tau) \in \mathcal{S}$, $[\![x]\!]^\sharp$ is a sound approximation of $[\![x]\!]^{\mathtt{ppt}}$.
Then, $v \in \gamma_\tau(\mathcal{A}, v^\sharp)$.

Therefore, to prove the soundness of the analysis, it is sufficient to prove that the abstract specifications of terms are sound approximation of the concrete specifications of terms.

6 Toward a Control Flow Analysis for λ-calculus

In this section we show how the abstract interpretation of Skel can be used to define a Control Flow Analysis (CFA) from the Skeletal Semantics of λ-calculus. A Control Flow Analysis computes an approximation of the Control Flow for higher-order languages. We aim at defining an analysis similar to 0-CFA for λ-calculus. Let t be a λ-term, the result of a 0-CFA for term t is two maps. The first one maps variables of t to an approximation of the values the variable can be bound to. The second one maps sub-terms of t to an approximation of the results of the evaluation of the sub-term.

In this section, we call **prg** the λ-term to be analyzed.

6.1 The State of the Abstract Interpretation

For our analysis, the state of the abstract interpretation contains two maps that we call C and ρ:

$$\begin{array}{l} \mathcal{A} \cdot C : \mathtt{ppt} \to \mathcal{P}_f(V^\sharp_{\mathbf{prg}}(ident) \times V^\sharp_{\mathbf{prg}}(lterm)) \\ \mathcal{A} \cdot \rho : \mathtt{ppt} \to V^\sharp_{\mathbf{prg}}(ident) \to \mathcal{P}_f(V^\sharp_{\mathbf{prg}}(ident) \times V^\sharp_{\mathbf{prg}}(lterm)) \end{array}$$

The C function maps a program point to an approximation of the result of evaluation of the sub-term of **prg** at the program point. For a program point pp, $C(\mathrm{pp})$ is a set of pairs (x, t), that are really λ-abstractions $\lambda x.t$.

ρ is a function from program points to some abstraction of an environment of λ-calculus. In classic 0-CFA, there is only one global environment, whereas in our analysis, there is one environment per program point. Practically, it means that to evaluate term t, a subterm of the main program **prg** at program point pp, we use the abstract environment $\mathcal{A} \cdot \rho(\mathrm{pp})$.

6.2 Specification of the Unspecified Types and Unspecified Terms

As for the big-step semantics with program points, the λ-terms are program types: $\mathcal{T} = \{\,\texttt{lterm}\,\}$.

We give the specifications of the unspecified types.

$$\frac{id \in \{\,x, y, z...\,\}}{\vdash^\sharp id \in V^\sharp_{\mathbf{prg}}(ident)}\ \textsc{Ident} \qquad\qquad \frac{\mathrm{pp} \in \texttt{ppt}}{\vdash^\sharp \mathrm{pp} \in V^\sharp_{\mathbf{prg}}(env)}\ \textsc{Env}$$

$V^\sharp_{\mathbf{prg}}(ident)$ is a set of variables. An abstract environment $\mathrm{pp}_e \in V^\sharp_{\mathbf{prg}}(env)$ is a program point and refers to $\mathcal{A}\cdot\rho(\mathrm{pp}_e)$ which maps variables to a set of pairs representing λ-abstractions. The definitions of the unspecified terms are:

$$[\![getEnv]\!]^\sharp(\mathcal{A}, x, \mathrm{pp}_e) = \mathcal{A}, Clos(\{\,(y, \mathrm{pp}\cdot 1, \mathrm{pp}) \mid (y, \mathrm{pp}\cdot 1) \in \mathcal{A}\cdot\rho(\mathrm{pp}_e)(x)\,\})$$
$$[\![extEnv]\!]^\sharp(\mathcal{A}, \mathrm{pp}_e, x, c) = \{\,\mathcal{A}\ \mathbf{with}\ \rho = \rho[\mathrm{pp}_v \to \mathcal{A}\cdot\rho(\mathrm{pp}_e)[x \to \mathcal{A}\cdot\rho(\mathrm{pp}_e)(x) \cup c]]\,\}, \mathrm{pp}_v \quad \mathrm{pp}_v \text{ fresh}$$

$\mathcal{A}\cdot\rho(\mathrm{pp}_e)(x)$ contains pairs of the form $(y, \mathrm{pp}\cdot 1)$ representing λ-abstractions, therefore $\mathbf{prg}@\,\mathrm{pp} = Lam(y, t)$. t has a program point that ends with 1 because it is the second element of the *Lam* constructor. getEnv type constraint the result to be in $V^\sharp_{\mathbf{prg}}(\texttt{clos})$, so we need to attach an environment to our pairs $(y, \mathrm{pp}\cdot 1)$. Because $(y, \mathrm{pp}\cdot 1)$ denotes a λ-abstraction defined at program point pp, the associated environment is pp.

$[\![extEnv]\!]^\sharp(\mathcal{A}, \mathrm{pp}_e, x, c)$ does two things: it modifies the state of the interpretation, and returns a *virtual* program point. Indeed, extEnv returns a new environment, therefore a program point, but what program point should be returned? We do not know until we evaluate a new `lterm` with this environment. Therefore, we create virtual program point that will be linked to a real program point later. The $[\![extEnv]\!]^\sharp$ modifies the abstract state such that $\rho(\mathrm{pp}_v)$ contains $\rho(\mathrm{pp}_e)$ plus the following constraint: $\rho(\mathrm{pp}_v)(x) = \rho(\mathrm{pp}_e(x)) \cup c$.

There remains to define our update functions:

$$\mathrm{update}^{in}_{eval}(\mathcal{A}, [\mathrm{pp}_e, \mathrm{pp}_t]) = \{\,\mathcal{A}\ \mathbf{with}\ \rho = \mathcal{A}\cdot\rho[\mathrm{pp}_t \to \mathcal{A}\cdot\rho(\mathrm{pp}_t) \sqcup^\sharp \mathcal{A}\cdot\rho(\mathrm{pp}_e)]\,\}, [\mathrm{pp}_t, \mathrm{pp}_t]$$
$$\mathrm{update}^{out}_{eval}(\mathcal{A}, [\mathrm{pp}_e, \mathrm{pp}_t], c) = \{\,\mathcal{A}\ \mathbf{with}\ C = C[\mathrm{pp}_t \to \mathcal{A}\cdot C(\mathrm{pp}) \cup c]\,\}, c$$

The first update function is used before performing the call *eval* pp_e pp_t, and the second update function is used after the call was done.

pp_e has type env and therefore is a program point which refers to $\mathcal{A}\cdot\rho(\mathrm{pp}_e)$. pp_t has type lterm, and therefore is also a program point denoting the sub-term of **prg** at pp_t. Because we want to compute the sub-term at program point pp_t, we should use the abstract environment $\mathcal{A}\cdot\rho(\mathrm{pp}_t)$. Therefore, the abstract state needs to be modified such that $\mathcal{A}\cdot\rho(\mathrm{pp}_e)$ is included into $\mathcal{A}\cdot\rho(\mathrm{pp}_t)$, and this is what the first update function does.

The second update function is called after the call is performed. The only thing does is to record the result into the map $\mathcal{A}\cdot C$.

A derivation gives a CFA:

$$\epsilon, \mathcal{A}_0, \{\,e \mapsto [\,], l \mapsto \underline{\epsilon}\,\}, eval\ e\ l \Downarrow^\sharp \mathcal{A}, v^\sharp$$

The derivation is implicitly parameterized by the λ-term **prg**. In the initial environment, l is mapped to the program point $\underline{\epsilon}$, the root of the λ-term **prg**. e is mapped to an empty

environment of λ-calculus. $\mathcal{A}_0$ is the initial state of the abstract interpretation with empty mappings $\mathcal{A}_0 \cdot \rho$ and $\mathcal{A}_0 \cdot C$. The result of the derivation is a value $v^\sharp$, but most importantly a new state of the interpretation $\mathcal{A}$, that contains mappings $\mathcal{A} \cdot \rho$ and $\mathcal{A} \cdot C$ that are the results of the CFA: $\mathcal{A} \cdot \rho$ gives an abstraction of what closures the variables in **prg** can be bound to, and $\mathcal{A} \cdot C$ gives an abstraction of the closures that can appear at a given program point.

7 Implementation

This work resulted in an implementation of an Abstract Interpreter Generator [8]: given a skeletal semantics, a specification of unspecified types and terms, it generates an abstract interpreter. We have used it to try our CFA analysis defined in previous sections, and we experimentally compared its output to another CFA program and we found no differences on the examples we tested. So far, our CFA has not been proven incorrect but it remains to do a formal proof or correction.

Moreover, the Abstract Interpreter Generator has been used to generate an abstract interpreter for a small imperative language, with basic integer arithmetic, conditional branchings and loops. For instance, one can do a basic interval analysis. However, we did not do relational analyses.

8 Conclusion

We presented our work to generate a CFA analysis for λ-calculus from a skeletal semantics. We presented what are Skeletal Semantics and how they can be interpreted to define a semantics. The strength of this approach is that one only needs to define the unspecified types and terms to mechanize a big-step semantics for a language that has a Skeletal Semantics. An example of how the big-step semantics of λ-calculus could be mechanized using Skeletal Semantics was given using the big-step semantics of Skel. Then we presented our first contribution which is a big-step semantics of Skel with program points. Our second contribution which is an abstract interpretation of Skel, and we stated a theorem of correction. The proof is a work in progress. Our final contribution is a CFA analysis built using the abstract interpretation of Skel for λ-calculus. We implemented an abstract interpreter generator that produces an analyzer from a Skeletal Semantics. We used this program to generate a CFA analyzer for λ-calculus. Our analyzer was experimentally tested and gives similar results to other 0-CFA analyzers.

However, when mechanizing a semantics using an intermediate language like Skel, we expect to lose precision compared to a language specific semantics. Moreover, there are several possible definitions of a Skeletal Semantics for given a language, the abstract interpretation precision may depend on the definition of the Skeletal Semantics. It is a language-independent approach to generates an abstract interpretation for languages with a Skeletal Semantics.

The proof of correction between the big-step semantics and the abstract interpretation must be mechanized. Moreover, our CFA analysis must be compared to other CFA analyses, in particular 0-CFA analysis and verify if we are systematically at least as precise as 0-CFA. Finally, there are not proofs of termination of the abstract interpretation, and this should be addressed.

References

[1] Bodin, M., Gardner, P., Jensen, T., Schmitt, A.: Skeletal semantics and their interpretations. Proceedings of the ACM on Programming Languages **3**(POPL), 1–31 (2019)

[2] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: NASA Formal Methods Symposium. pp. 3–11. Springer (2015)

[3] Cousot, P., Cousot, R.: 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. pp. 238–252 (1977)

[4] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astrée analyzer. In: European Symposium on Programming. pp. 21–30. Springer (2005)

[5] Noizet, L.: Necro Library, https://gitlab.inria.fr/skeletons/necro, https://gitlab.inria.fr/skeletons/necro

[6] Noizet, L., Schmitt, A.: Semantics in Skel and Necro. In: ICTCS 2022 - Italian Conference on Theoretical Computer Science. CEUR Workshop Proceedings, Rome, Italy (Sep 2022)

[7] Roșu, G., Șerbănuță, T.F.: An overview of the k semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010)

[8] Rébiscoul, V.: Abstract Interpreter Generator, https://gitlab.inria.fr/skeletons/abstract-interpreter-generator

[9] Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. In: LISP AND SYMBOLIC COMPUTATION. pp. 288–298 (1993)

[10] Schmidt, D.A.: Natural-semantics-based abstract interpretation (preliminary version). In: International Static Analysis Symposium. pp. 1–18. Springer (1995)

A Skeletal Semantics of λ-calculus

```
type ident
type env

type clos =
| Clos (ident, lterm, env)

type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)

val extEnv : (env, ident, clos) → env
val getEnv : (ident, env) → clos

val eval (s:env) (l:lterm): clos =
        branch
                let Lam (x, t) = l in
                Clos (x, t, s)
        or
                let Var x = l in
                getEnv (x, s)
```

```
or
        let App (t1, t2) = l in
        let Clos (x, t, s') = eval s t1 in
        let w = eval s t2 in
        let s'' = extEnv (s', x, w) in
        eval s'' t
end
```

B Typing Rules of Skeletons and Terms

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\textbf{val}\ x : \tau[= t] \in \mathcal{S}}{\Gamma \vdash x : \tau}\ \textsc{TermDef} \qquad \frac{\Gamma \vdash t : \tau \qquad \mathrm{C} : (\tau, \tau')}{\Gamma \vdash \mathrm{C}\, t : \tau'}\ \textsc{Const}$$

$$\frac{\forall i, \Gamma \vdash t_i : \tau_i}{\Gamma \vdash (t_1, \ldots, t_n) : (\tau_1, \ldots, \tau_n)}\ \textsc{Tuple} \qquad \frac{\Gamma + p \leftarrow \tau \vdash S : \tau'}{\Gamma \vdash (\lambda p : \tau \to S) : \tau \to \tau'}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash S_1 : \tau \qquad \ldots \qquad \Gamma \vdash S_n : \tau}{\Gamma \vdash (S_1 \ldots S_n) : \tau}\ \textsc{Branch} \qquad \frac{\Gamma \vdash S : \tau \qquad \Gamma + p \leftarrow \tau \vdash S' : \tau'}{\Gamma \vdash \mathrm{let}\, p = S \,\mathrm{in}\, S' : \tau'}\ \textsc{LetIn}$$

$$\frac{\Gamma \vdash t_0 : \tau_1 \to \cdots \to \tau_n \to \tau \qquad \forall i\ \Gamma \vdash t_i : \tau_i}{\Gamma \vdash (t_0\ t_1 \ldots t_n) : \tau}\ \textsc{App}$$

C The Functions of a Skeletal Semantics $\mathcal{S}$

$$\mathrm{Funs}(\Gamma, \textbf{let}\ p\ =\ S_1\ \textbf{in}\ S_2) = \mathrm{Funs}(\Gamma, S_1) \cup \mathrm{Funs}(\Gamma + p \leftarrow \tau, S_2)$$

$$\mathrm{Funs}(\Gamma, \textbf{branch}\ S_1 \ldots S_n\ \textbf{end}) = \bigcup_{i=1}^{n} \mathrm{Funs}(\Gamma, S_i) \qquad \mathrm{Funs}(\Gamma, t_0\ t_1 \ldots t_n) = \bigcup_{i=0}^{n} \mathrm{Funs}(\Gamma, t_i)$$

$$\mathrm{Funs}(\Gamma, \lambda p : \tau \to S_0) = \{\, \Gamma, \lambda p : \tau \to S_0 \,\} \cup \mathrm{Funs}(\Gamma + p \leftarrow \tau, S_0)$$

$$\mathrm{Funs}(\Gamma, (t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \mathrm{Funs}(\Gamma, t_i) \qquad \mathrm{Funs}(\Gamma, C\, t) = \mathrm{Funs}(\Gamma, t) \qquad \mathrm{Funs}(\Gamma, x) = \emptyset$$

The set of λ-abstractions in the Skeletal Semantics $\mathcal{S}$ is:

$$\mathrm{Funs}(\mathcal{S}) \equiv \bigcup_{\textbf{val}\ x:\tau = t \in \mathcal{S}} \mathrm{Funs}(\varnothing, t)$$

D Correction of Big-Step Interpretation with Program Points

Definition 2. To relate element with or without program points, we define the following function γ, such that $\forall \tau, \gamma_\tau \in V^{\texttt{ppt}}_{\textbf{prg}}(\tau) \to V(\tau)$. The function γ is parameterized by the global value of type $\tau \in \mathcal{T}$ and satisfies the following constraints.

- $\gamma_\tau(\text{pp}) = \textbf{prg}@\,\text{pp}$ and $\tau \in \mathcal{T}$
- $\gamma_{\tau_1 \times \cdots \times \tau_n}((v'_1, \ldots, v'_n)) = (\gamma_{\tau_1}(v'_1), \ldots, \gamma_{\tau_n}(v'_n))$
- $\gamma_{\tau_a}(C\ v') = C\ \gamma_\tau(v')$ with $C : (\tau, \tau_a)$
- Suppose $\Gamma \vdash E$, and $\Gamma \vdash E'$
  $\gamma_{env}(E') = E \iff \text{dom } E' = \text{dom } E \wedge \forall x \in \text{dom } E', \gamma_{\Gamma(x)}(E'(x)) = E(x)$
- $\gamma_{\tau_1 \to \tau_2}((p, S, E')) = (p, S, \gamma_{env}(E'))$
- Suppose $\textbf{val}\ f : \tau_1 \to \cdots \to \tau_n \to \tau[= t] \in \mathcal{S}$, then
  $\gamma_\tau((f, [v'_1, \ldots, v'_n], k)) = (f, [\gamma_{\tau_1}(v'_1), \ldots, \gamma_{\tau_n}(v'_n)], k)$

Definition 3. Take f such that $\textbf{val}\ f : \tau_1 \to \cdots \to \tau_n \to \tau \in \mathcal{S}$:

$$\gamma_{unspec}([\![f]\!]^{\texttt{ppt}}) = [\![f]\!] \iff \forall (v'_i, v_i) \in V^{\texttt{ppt}}_{\textbf{prg}}(\tau_i) \times V(\tau_i) \text{ such that } \gamma_{\tau_i}(v'_i) v_i \\ \gamma_\tau([\![f]\!]^{\texttt{ppt}}(v'_1, \ldots, v'_n)) = [\![f]\!](v_1, \ldots, v_n)$$

Theorem 2. Let $\gamma_{env}(E') = E$, and suppose for all unspecified functions f, $\gamma_{unspec}([\![f]\!]^{\texttt{ppt}}) = [\![f]\!]$, then:

$$E, S \Downarrow v \implies \exists v', \quad E', S \Downarrow^{PP} v' \text{ and } \gamma_\tau(v') = v$$