Chameleon: The Gist of Dynamic Programming Languages

Samuele Buro
Dept. of Computer Science
University of Verona
Email: samuele.buro@univr.it

Michele Pasqua
Dept. of Computer Science
University of Verona
Email: michele.pasqua@univr.it

Isabella Mastroeni
Dept. of Computer Science
University of Verona
Email: isabella.mastroeni@univr.it

***Abstract*—Dynamic programming languages, such as JavaScript and PHP, are widespread and heavily used. They provide very useful "dynamic" features, like run-time type inference, dynamic method calls, and built-in dynamic data structures. This makes it hard to build static analyzers, for automatic errors discovery. Yet, exploiting harmful behaviors in such programs, especially in web applications, can have significant impacts. In this paper, we present Chameleon, a core programming language summarizing the main features of the dynamic programming paradigm. Chameleon can be useful in defining, testing and comparing static analyses, aiming at preventing bugs and errors in programs written in dynamic programming languages. With Chameleon, static analysis experts could define and test control mechanisms without the burden to take in consideration the technical details characterizing a specific real-world programming language.**

Keywords–Programming language design; Dynamic programming languages; Program static analysis.

I. INTRODUCTION

In the last years, dynamic programming languages, such as JavaScript or PHP, exponentially enhanced their popularity and nowadays are deeply used in a very wide range of applications. For instance, JavaScript is the *de facto* standard for client-side web programming, while, on the server-side, PHP, Python, and Ruby are the most common used languages. This success is mainly due to the several features that such languages provide to developers, making the writing of programs easier and faster.

Although there is no black and white distinction between static and dynamic programming languages, the latter basically follows two main paradigms: The first, justifying also the adjective *dynamic*, is the lack of a *static* type system. Dynamic programming languages still have types, but they are checked at run-time, rather than compile-time. The absence of strict static checks promotes the second aspect of dynamic languages, namely a greater flexibility at run-time. The basic idea is that operations which may be statically forbidden or not expressible in other languages should be allowed and given some semantics, and the program execution should continue whenever possible.

The benefit of these design choices is that programmers have a high flexibility in writing code. The downside is that errors occur at run-time and little or no information is available for developer tools to prevent these errors statically. A static analyzer is a tool which abstractly executes the program, *i.e.,* approximates all its possible behaviors. The computed approximation is then used to detect bugs or to provide useful information to developer tools. Examples of static analysis comprise data-flow analysis [1], invariants analysis [2] and model-checking [3].

Due to the dynamic nature of these languages, it is, indeed, very hard for static analysis experts to develop such control mechanisms. In addition to the aforementioned issues, there is the *heterogeneity* problem: it is not necessarily the case that an analysis designed for a programming language works also for the others.

Many authors [4]–[6] define their own toy language, in order to present the analysis they are introducing or improving. This is surely a burden for authors and, more importantly, it does not allow comparisons between similar static analyses, since the underling language is different.

To overcome these issues, we propose a core programming language, called *Chameleon* (also typesetted as ehameleon), summarizing the main features of dynamic programming languages. It abstracts the implementation details characterizing each language, allowing to focus on the analysis of the dynamic features only. Indeed, building static analyzers for real programming languages is a very complex and time-consuming engineering task. Having a simple language, yet sufficiently expressive to model the main dynamic features, allows to define new static analyses faster and easier.

Furthermore, Chameleon could be used as a common ground for the definition and comparison, of static analysis techniques, aiming at reasoning about programs written in dynamic programming languages, without being restricted to a particular language. Ideally, when an analysis has been sufficiently tested on Chameleon, then it could be ported to the target *real-world* programming language with just engineering efforts and without losing theoretical solidity.

Outline: In Section II we describe the Chameleon language, first its syntax (Subsection II-A) and then its semantics (Subsection II-B). Finally, we draw conclusions, in Section III.

II. THE EHAMELEON LANGUAGE

Chameleon is a programming language designed specifically to ease the definition of static analyses, with the focus on dynamic features of programming languages. Its core consists in a classic imperative language with assignments, conditionals and iterative constructs. This latter is extended with *functions/procedures* and with *non-determinism*. Non-determinism is modeled by means of an input construct, allowing programs to receive input values during execution. Chameleon is equipped with standard basic values (booleans, integers, rationals and strings), as well as inductive data-structures, such as *finite lists* of values and *finite dictionaries* (*i.e.,* identifier-value associations).

 ISBN: 978-1-61208-755-9

⟨*prog*⟩ ::= $\overline{\langle \textit{fundef} \rangle}_{;}$ ⟨*com*⟩
⟨*fundef*⟩ ::= `function` ⟨*id*⟩ `(` $\overline{\langle \textit{exp} \rangle}_{,}$ `)` `{` ⟨*com*⟩ `}`
⟨*com*⟩ ::=
| `skip`
| ⟨*id*⟩ `:=` ⟨*exp*⟩
| ⟨*id*⟩ `[` ⟨*exp*⟩ `]` `:=` ⟨*exp*⟩
| `if` ⟨*exp*⟩ `then` `{` ⟨*com*⟩ `}` `else` `{` ⟨*com*⟩ `}`
| `while` ⟨*exp*⟩ `do` `{` ⟨*com*⟩ `}`
| ⟨*com*⟩ `;` ⟨*com*⟩
| `return` ⟨*exp*⟩
| ⟨*exp*⟩
⟨*exp*⟩ ::=
| `(` ⟨*exp*⟩ `)`
| ⟨*value*⟩
| ⟨*id*⟩
| ⟨*id*⟩ `(` $\overline{\langle \textit{value} \rangle}_{,}$ `)`
| ⟨*exp*⟩ `[` ⟨*exp*⟩ `]`
| `(` ⟨*type*⟩ `)` ⟨*exp*⟩
| `eval` ⟨*exp*⟩
| `input()`
| `size` `(` ⟨*exp*⟩ `)`
| `concat` `(` ⟨*exp*⟩ `,` ⟨*exp*⟩ `)`
| `charat` `(` ⟨*exp*⟩ `,` ⟨*exp*⟩ `)`
| `substr` `(` ⟨*exp*⟩ `,` ⟨*exp*⟩ `,` ⟨*exp*⟩ `)`
| `not` ⟨*exp*⟩
| `-` `(` ⟨*exp*⟩ `)`
| ⟨*exp*⟩ ⟨*bop*⟩ ⟨*exp*⟩
⟨*bop*⟩ ::= `*` | `/` | `+` | `-` | `<=` | `>=` | `<` | `>` | `==` | `not` | `and` | `or`
⟨*type*⟩ ::= `bool` | `int` | `rat` | `str`
| `[` `]`
| `[` $\overline{\langle \textit{type} \rangle}_{,}$ `]`
| `[` $\overline{\langle \textit{id} : \textit{type} \rangle}_{,}$ `]`
⟨*value*⟩ ::=
| $\bot$
| $b \in \mathbb{B}$
| $i \in \mathbb{Z}$
| $q \in \mathbb{Q}$
| $s \in \mathbb{S}$
| ⟨*collection*⟩
⟨*collection*⟩ ::= `[` `]` | `[` $\overline{\langle \textit{exp} \rangle}_{,}$ `]` | `[` $\overline{\langle \textit{id} : \textit{exp} \rangle}_{,}$ `]`
⟨*id*⟩ ::= $x \in \mathbb{X}$

Figure 1. The syntax of Chameleon

Concerning properly dynamic features, Chameleon is *not statically typed* and it applies *type coercion* when needed. The language has a (limited) reflection mechanism, implemented with an *eval* construct. Finally, Chameleon expressions can have *side-effects*.

A. Syntax

The syntax of Chameleon is specified by the context-free grammar depicted in Figure 1. A *program* $P \in \langle \textit{prog} \rangle$ is a list of *function definitions* $f \in \overline{\langle \textit{fundef} \rangle}_{;}$ followed by a *command* $c \in \langle \textit{com} \rangle$, which in turn can be a standard statement of an imperative language (like, in order, the do-nothing command, the assignment of a variable, a list element, or a field, the conditional statement, the conditional loop, and the composition), a return statement (which can be used either to leave the execution of a function with a value or to terminate the program when employed outside of functions body), or an *expression* $e \in \langle \textit{exp} \rangle$.

An *expression* e in Chameleon is inductively defined by the syntactic category ⟨*exp*⟩. Its smallest building block are *identifiers* $x \in \langle \textit{id} \rangle = \mathbb{X} = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}^*$, and *simple values* which consist of the undefined value $\bot$, the *booleans* $b \in \mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$, the *integers* $i \in \mathbb{Z}$, the *floats* $q \in \mathbb{Q}$, the *strings* $s \in \mathbb{S}$ (a string of the language is a sequence of characters over the alphanumeric alphabet enclosed by double quotes, *e.g.,* , "`foo`", "`bar`", etc., and we assume the Java-like syntax for characters escaping), and the empty list `[]`. Compound expressions are built inductively: If e is an expression, then `(`e`)` is a parenthesized expression; If $e_0, \ldots, e_n$ and $x_0, \ldots, x_n$ are a sequence of expressions and a sequence of identifiers, respectively, then `[`e_0`, ...,`e_n`]` is a non-empty list and `[`x_0 `:` e_0`, ...,`x_n `:` e_n`]` is a dictionary; If $f \in \mathbb{X}$ is a function name, then f`(`e_0`, ...,`e_n`)` is a function call with actual parameters $e_0, \ldots, e_n \in \langle \textit{exp} \rangle$, whereas f`()` is a function call with no arguments; If $c \in \langle \textit{collection} \rangle$ is a list or a dictionary (*i.e.,* a *collection*), then c`[`e`]` is the access of an element in c, namely, a list access or a field access depending on the nature of c; If t is a type, then `(`t`)` e is a cast to the type t. The rest of the rules of the grammar are self-explanatory and their purpose can be easily recovered by the semantic rules given in the next section. Briefly, they include the `eval` statement, the `input()` function, and several operators for the most common operations between values (the precedence rules for the binary operators in ⟨*bop*⟩ are the standard ones, and the associativity is always left to right).

In the following, we refer to the *set of all terms* $\mathcal{T}$ defined as the union of the sets of terms generated by each syntactic category of the Chameleon grammar.

B. Semantics

In this section, we formally describe the operational semantics of Chameleon. We start by defining the concepts of *ground values*, *types*, and *state* during an arbitrary step of computation, then we provide a *small-step operational semantics*.

1) Ground Values and Types: Let $\mathbb{V}$ be the set of *ground values* (with metavariable v) inductively defined as the smallest set such that $\{\bot\} \cup \mathbb{B} \cup \mathbb{Z} \cup \mathbb{Q} \cup \mathbb{S} \cup \{\mathtt{[]}\} \subseteq \mathbb{V}$, and if $v_0, \ldots, v_n \in \mathbb{V}$ and $x_0, \ldots, x_n \in \mathbb{X}$, then `[`$v_0$`, ...,`$v_n$`]` and `[`$x_0$ `:` v_0`, ...,`x_n `:` v_n`]` belong to $\mathbb{V}$. Moreover, we define the set $\mathbb{C} = \mathbb{L} \cup \mathbb{D}$ of *ground collections*, where $\mathbb{L} = \{\, l \in \mathbb{V} \mid l = \mathtt{[}v_0, \ldots, v_n\mathtt{]} \,\} \cup \{\mathtt{[]}\}$ of *ground lists* and the set $\mathbb{D} = \{\, d \in \mathbb{V} \mid d = \mathtt{[}x_0 : v_0, \ldots, x_n : v_n\mathtt{]} \,\}$ of *ground dictionaries*.

The type of a value is inductively defined by the function $\tau \colon \mathbb{V} \to \langle \textit{type} \rangle_\bot$ as follows (see the previous section for the meaning of the metavariables employed in the definition): $\tau(\bot) = \bot$, $\tau(b) = \mathtt{bool}$, $\tau(i) = \mathtt{int}$, $\tau(f) = \mathtt{rat}$, and $\tau(s) = \mathtt{str}$. Moreover, if $l = \mathtt{[}v_1, \ldots, v_n\mathtt{]}$ is a (potentially empty) ground list, then $\tau(l) = \mathtt{[}\tau(v_1), \ldots, \tau(v_n)\mathtt{]}$ is the type of l, and if $d = \mathtt{[}x_0 : v_0, \ldots, x_n : v_n\mathtt{]}$ is a ground dictionary, then $\tau(d) = \mathtt{[}x_0 : \tau(v_0), \ldots, x_n : \tau(v_n)\mathtt{]}$ is the type of d. If $t, t' \in \tau(\mathbb{D})$, we define the equivalence relation $t \sim t'$ if and only if t' is a permutation of t.

In the following, we refer to $\bot$, `bool`, `int`, `rat`, and `str` as *simple types*, and to the other as *compound types*. Moreover, if

$t = [t_0, \ldots, t_n]$ or $t = [x_0 : t_0, \ldots, x_n : t_n]$ is a compound type, we define the length of t as $|t| = n+1$, and if $t = []$ then $|t| = 0$. Finally, we define the partial order relation $\preccurlyeq$ between types: $\bot \preccurlyeq t \preccurlyeq t$ for each type t, and $\texttt{bool} \preccurlyeq \texttt{int} \preccurlyeq \texttt{rat} \preccurlyeq \texttt{str}$; if $t = [t_1, \ldots, t_n]$ and $t' = [t'_0, \ldots, t'_n]$, then $t \preccurlyeq t'$ if and only if $t_i \preccurlyeq t'_i$ for each $i = 0 \ldots n$; If $t = [x_0 : t_0, \ldots, x_n : t_n]$ and $t' = [x_0 : t'_0, \ldots, x_n : t'_n]$, then $t \preccurlyeq t'$ if and only if $t_i \preccurlyeq t'_i$ for each $i = 0 \ldots n$ or there are $\hat{t}$ and $\hat{t}'$ such that $t \sim \hat{t} \preccurlyeq \hat{t}' \sim t'$.

2) State: Let $\Sigma = \mathbb{X} \to \mathbb{V}$ be the set of *environments*, and let $\mathrm{P} = \mathbb{X} \to \left(\bigcup_{n \in \mathbb{N}} \mathbb{X}^n \times \langle com \rangle\right)_\bot$ be the set of *function definitions maps*. The *state* of the language during an arbitrary step of computation is an element of the set $\Pi = \Sigma^+ \times \Sigma \times \mathrm{P}$. Given $\pi = (\dot{\sigma}, \gamma, \rho) \in \Pi$, where $\dot{\sigma} = \sigma_0 \ldots \sigma_n$, the second component γ denotes the *global state* (*i.e.,* the variables accessible throughout the program, unless shadowed by local ones), and $\dot{\sigma}$ is a non-empty list of *local states*. Intuitively, the list $\dot{\sigma}$ of local states shall be used to handle the scope of variables during a chain of function calls: Every time a function is called, a new component $\sigma \in \Sigma$ is appended to $\dot{\sigma}$, and new bindings between formal and actual parameters are created in σ. Conversely, every time a `return` statement is reached, the last component σ_n of $\dot{\sigma}$ is dropped, and the previous bindings are automatically restored. Moreover, the third component ρ in π serves to keep track of all the functions declared by a program. For instance, the following function

```
function factorial(n) {
    if (n < 2) then { return 1 };
    return n * factorial(n - 1)
}
```

is stored in ρ as $\rho(\texttt{factorial}) = (\texttt{n}, c)$, where c is the body of the function.

In order to handle the state in the small-step rules of the language, we define some compact notations that will be used throughout the paper: We write $\sigma = \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \}$ to define the environment $\sigma \in \mathbb{X} \to \mathbb{V}$ such that $\sigma(x) = v_i$ if $x = x_i$ for some $i = 1 \ldots n$ and $\sigma(x) = \bot$ otherwise. Note that, unlike dictionaries, environments can be infinite objects as well as completely undefined (*i.e.,* when $n = 0$ and therefore $\sigma = \{\ \}$, then $\sigma(x) = \bot$ for all identifiers x). Moreover, if $\sigma \in \Sigma$, we denote the *update* of the variable x in σ with a value v as the new environment $\sigma[x \leftarrowtail v]$ defined as

$$\sigma[x \leftarrowtail v](y) = \begin{cases} v & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

and we naturally extend the notation to an arbitrary number of variables, *i.e.,* $\sigma[x_0 \leftarrowtail v_0, \ldots, x_n \leftarrowtail v_n] = (\cdots(\sigma[x_0 \leftarrowtail v_0]) \cdots)[x_n \leftarrowtail v_n]$. We also extend this notation to a function definition map ρ.

Finally, given a state $\pi = (\dot{\sigma}, \gamma, \rho)$ where $\dot{\sigma} = \sigma_0 \ldots \sigma_n$, the *appending* of a new environment σ to $\dot{\sigma}$ is defined by $\pi \triangleleft \sigma = (\dot{\sigma} \triangleleft \sigma, \gamma, \rho) = (\sigma_0 \ldots \sigma_n \sigma, \gamma, \rho)$, whereas the *dropping* of the last component of $\dot{\sigma}$ is denoted by $\pi_\downarrow = (\dot{\sigma}_\downarrow, \gamma, \rho) = (\sigma_0 \ldots \sigma_{n-1}, \gamma, \rho)$ if $n > 0$. The *access* to the value of a variable x in π is defined by

$$\pi(x) = \begin{cases} \sigma_n(x) & \text{if } \sigma_n(x) \neq \bot \\ \gamma(x) & \text{otherwise} \end{cases}$$

This last definition actually formalizes the *shadowing* of global variables by local ones.

3) Operational Semantics: Given the above definitions, we provide the *small-step operational semantics* à la Plotkin [7]. In particular, we define the binary relation $\to$ over the set $\Pi \times \mathcal{T}$ accordingly to the inference rules provided in the next sections. Because of space limitations, we do not describe the semantics of the standard operations of the language (for which we redirect the reader to [8]), but we only focus on the key features of Chameleon.

Semantics of Function Calls: In order to compute the resulting value of an arbitrary function call $f(e_1, \ldots, e_n)$ in a computational state $\pi = (\dot{\sigma}, \gamma, \rho)$, Chameleon implements a call-by-value strategy: Firstly, all actual parameters $e_0, \ldots, e_n$ are evaluated to a ground value (rule FunCall). Then, if the function f is undefined in ρ, the undefined value is returned (rule FunCall-B1), otherwise a new environment in which the bindings between actual and formal parameters are defined is appended to $\dot{\sigma}$, and the computation continues from the function body (rules FunCall-B2 and FunCall-B3).

Semantics of Type Casting: The explicit type conversion $(t)\ e$ allows programmers to change the value of the expression e from its original type to the new type t. For each type t, we define a conversion function $\hookrightarrow_t$ that implements the type cast policy. More precisely, $\hookrightarrow_t : \mathbb{V} \to \mathbb{V}_t$ moves values from $\mathbb{V}$ to $\mathbb{V}_t = \{ v \in \mathbb{V} \mid \tau(v) = t \}$, accordingly to the type of the input value, namely $\hookrightarrow_t (v) = \hookrightarrow_{t,\tau(v)} (v)$ where $\hookrightarrow_{t,\tau(v)} : \mathbb{V}_t \to \mathbb{V}_{\tau(v)}$. Given two types t and t', the definition of these functions is the standard conversion if t and t' are simple types, the identity function if $t = t'$, and otherwise $\hookrightarrow_{t,t'} (v)$ is inductively defined as follows:

$$\begin{cases} \hookrightarrow_{t,t'} (v) = \bot & \text{if } t \text{ or } t' \text{ is a simple type} \\ \hookrightarrow_{t,t'} (v) = \bot & \text{if } t, t' \notin \tau(\mathbb{L}) \text{ or } t, t' \notin \tau(\mathbb{D}) \\ \hookrightarrow_{t,t'} (v) = \bot & \text{if } |t| \neq |t'| \\ \hookrightarrow_{t,t'} ([v_1, \ldots, v_n]) = [\ \hookrightarrow_{t'_1} (v_1), \ldots, \hookrightarrow_{t'_n} (v_n)] & \\ \quad \text{if } t' = [t'_0, \ldots, t'_n] & \\ \hookrightarrow_{t,t'} ([x_0 : v_0, \ldots, x_n : v_n]) = & \\ \quad [x_0 : \hookrightarrow_{t'_0} (v_1), \ldots, x_n : \hookrightarrow_{t'_n} (v_n)] & \\ \quad \text{if } t' \sim [x_0 : t'_0, \ldots, x_n : t'_n] & \end{cases}$$

For instance, concerning simple types, if $v =$ "`b4r`" then $\hookrightarrow_{\texttt{str},\texttt{int}} (v) = \texttt{4}$, or if $v = \texttt{0}$, then $\hookrightarrow_{\texttt{int},\texttt{bool}} (v) = \texttt{false}$, or if $v = \texttt{3.5}$, then $\hookrightarrow_{\texttt{int},\texttt{str}} (v) =$ "`3.5`", etc. Given these premises, the rules Cast and Cast-B are now self-explanatory.

Reflection and Non-Determinism: The `eval` e statement enables the runtime execution of Chameleon code dynamically crafted by programs. Rule Eval evaluates the expression e until a value v is obtained. If $v =$ "P" is a string representing a valid Chameleon program, then $\hat{P}$ (namely, the unescaped version of P) is executed in the state π (thus, allowing side effects, see Eval-B1). Otherwise, if v is not the representation of a valid program, $\bot$ is returned (Eval-B2).

On the other hand, the `input()` expression allows *unbounded non-determinism* [9]. The implementation of the `input()` statement requires the user to supply an input (*i.e.,* a string) before continuing the computation. From the trace semantics point of view, any $s \in \mathbb{S}$ is a possible outcome of these statement, as modeled by the rule Input.

Operations and Type Coercion: Chameleon employs a neat type coercion system in order to let values to transparently

$$\textsc{ParExp}\ \frac{\langle \pi, e\rangle \to \langle \pi', e'\rangle}{\langle \pi, (e)\rangle \to \langle \pi', (e')\rangle} \qquad \textsc{ParExp-B}\ \frac{-}{\langle \pi, (v)\rangle \to \langle \pi, v\rangle}$$

$$\textsc{Id}\ \frac{-}{\langle \pi, x\rangle \to \langle \pi, \pi(x)\rangle}$$

$$\textsc{FunCall}\ \frac{\langle \pi, e_{i+1}\rangle \to \langle \pi', e'_{i+1}\rangle}{\langle \pi, f(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n)\rangle \to \langle \pi', f(v_1, \ldots, v_i, e'_{i+1}, \ldots, e_n)\rangle}$$

$$\textsc{FunCall-B1}\ \frac{-}{\langle \pi, f(v_1, \ldots, v_n)\rangle \to \langle \pi, \bot\rangle}\ \rho(f) = \bot$$

$$\textsc{FunCall-B2}\ \frac{-}{\langle \pi, f(v_1, \ldots, v_n)\rangle \to \langle \pi \triangleleft \{\, x_1 \hookleftarrow v_1, \ldots, x_m \hookleftarrow v_m \,\}, c\rangle}\ \rho(f) = ((x_1, \ldots, x_m), c) \wedge n \geq m$$

$$\textsc{FunCall-B3}\ \frac{-}{\langle \pi, f(v_1, \ldots, v_n)\rangle \to \langle \pi \triangleleft \{\, x_1 \hookleftarrow v_1, \ldots, x_n \hookleftarrow v_n \,\}, c\rangle}\ \rho(f) = ((x_1, \ldots, x_m), c) \wedge n < m$$

$$\textsc{Cast}\ \frac{\langle \pi, e\rangle \to \langle \pi', e'\rangle}{\langle \pi, (t)\ e\rangle \to \langle \pi', (t)\ e'\rangle} \qquad \textsc{Cast-B}\ \frac{-}{\langle \pi, (t)\ v\rangle \to \langle \pi, \hookrightarrow_t (v)\rangle}$$

$$\textsc{Eval}\ \frac{\langle \pi, e\rangle \to \langle \pi', e'\rangle}{\langle \pi, \texttt{eval}\ e\rangle \to \langle \pi', \texttt{eval}\ e'\rangle} \qquad \textsc{Eval-B1}\ \frac{-}{\langle \pi, \texttt{eval}\ v\rangle \to \langle \pi, \hat{P}\rangle}\ \exists \hat{P} \in \langle \textit{prog}\rangle \,.\, v = \text{“}P\text{”}$$

$$\textsc{Eval-B2}\ \frac{-}{\langle \pi, \texttt{eval}\ v\rangle \to \langle \pi, \bot\rangle}\ \nexists \hat{P} \in \langle \textit{prog}\rangle \,.\, v = \text{“}P\text{”}$$

$$\textsc{Input}\ \frac{-}{\langle \pi, \texttt{input()}\rangle \to \langle \pi, s\rangle}\ \forall s \in \mathbb{S}$$

Figure 2. Small-step semantics of Chameleon expressions.

flow from one type to another when needed. Suppose that $\otimes \in \langle \textit{bop}\rangle$ is defined for integers and rationals and let us denote by $\otimes_{\texttt{int}} \colon \mathbb{V}^2_{\texttt{int}} \to \mathbb{V}_{\texttt{int}}$ and $\otimes_{\texttt{rat}} \colon \mathbb{V}^2_{\texttt{rat}} \to \mathbb{V}_{\texttt{rat}}$ the typed versions of $\otimes$. Consider the expression $v \otimes v'$ for two arbitrary ground values: The goal is to get a value $v'' = v \otimes v'$ in a way that depends only on the set of types on which $\otimes$ is defined. For instance, if $\otimes = *$, we want to provide a meaning to expressions like `(true * "a")`, `(5 * false)`, etc. Note that computing v'' is not a trivial task, especially when seeking a general method.

The strategy implemented in the Chameleon interpreter is based on the previously defined partial order $\preccurlyeq$ on types. The algorithm for the computation of v'' is described as follows:

1) We compute the set of types on which $\otimes$ is defined, namely $\mathrm{dom}(\otimes) = \{\, \bot \,\} \cup \{\, \texttt{int}, \texttt{rat} \,\} \cup \tau(\mathbb{L}) \cup \tau(\mathbb{D})$. By this definition, every operator is defined on the undefined type in a vacuous manner, and inductively on compound types. More precisely, this means that $\bot \otimes v = v \otimes \bot = \bot$, and if $v_0, v'_0 \ldots, v_n, v'_n$ is a sequence of values, then $\texttt{[}v_0, \ldots, v_n\texttt{]} \otimes \texttt{[}v'_0, \ldots, v'_n\texttt{]} = \texttt{[}v_0 \otimes v'_0, \ldots, v_n \otimes v'_n\texttt{]}$ and similarly for dictionary values;
2) We refine $\mathrm{dom}(\otimes)$ in order to get all the types greater than $\tau(v)$ or $\tau(v')$, namely $\mathrm{dom}_{\uparrow}(\otimes) = \{\, t \in \mathrm{dom}(\otimes) \mid t \geq \tau(v) \vee t \geq \tau(v') \,\}$, and the types lower than $\tau(v)$ or $\tau(v')$, namely $\mathrm{dom}_{\downarrow}(\otimes) = \{\, t \in \mathrm{dom}(\otimes) \mid t \leq \tau(v) \vee t \leq \tau(v') \,\}$;
3) We compute the least upper bound between (i) the greatest lower bound of $\mathrm{dom}_{\uparrow}(\otimes)$ and (ii) the least upper bound of $\mathrm{dom}_{\downarrow}(\otimes)$, namely $t = \bigvee\{\, \wedge \mathrm{dom}_{\uparrow}(\otimes), \vee \mathrm{dom}_{\downarrow}(\otimes) \,\}$;
4) The result of the computation is defined as $v'' = v \otimes_t v'$.

The rules for computing the result of a binary operation are now trivial. Firstly, we evaluate left-to-right the expressions in the operation until values are obtained:

$$\textsc{Exp-L}\ \frac{\langle \pi, e_1\rangle \to \langle \pi', e'_1\rangle}{\langle \pi, e_1 \otimes e_2\rangle \to \langle \pi', e'_1 \otimes e_2\rangle} \qquad \textsc{Exp-R}\ \frac{\langle \pi, e_2\rangle \to \langle \pi', e'_2\rangle}{\langle \pi, v \otimes e_2\rangle \to \langle \pi', v \otimes e'_2\rangle}$$

Then, we compute the result applying the algorithm described above:

$$\textsc{Exp-B}\ \frac{-}{\langle \pi, v \otimes v'\rangle \to \langle \pi', v''\rangle}$$

where $v'' = v \otimes_t v'$ and $t = \bigvee\{\, \wedge \mathrm{dom}_{\uparrow}(\otimes), \vee \mathrm{dom}_{\downarrow}(\otimes) \,\}$.

Commands and Return Statement: Since most of Chameleon commands are common to the majority of the imperative languages, we only discuss here the rule of the return statement, and we redirect the reader to [8] for a detailed explanation of the other commands.

When a `return` e statement is met, the expression e is evaluated in order to obtain a value v:

$$\textsc{Ret}\ \frac{\langle \pi, e\rangle \rightarrow \langle \pi', e'\rangle}{\langle \pi, \texttt{return}\ e\rangle \rightarrow \langle \pi', \texttt{return}\ e'\rangle}$$

Then, the following rule returns the value v and restores the bindings existing previously of the function call:

$$\textsc{Ret-B}\ \frac{-}{\langle \pi, \texttt{return}\ v\rangle \rightarrow \langle \pi_{\downarrow}, v\rangle}$$

III. Conclusion

In this paper, we have presented Chameleon, a minimal language capturing the main features of dynamic programming languages. In particular, it is an imperative non-deterministic language with functions/procedures and built-in inductive data-structures, such as finite lists and finite dictionaries. Concerning the dynamic features, Chameleon is not statically typed, with a mechanism for type coercion. It supports (limited) reflection, implemented by means of an eval-like construct, and expressions can have side-effects.

The aim of Chameleon is to provide a common ground for static analyses developers, in order to easily define and test their control mechanisms. To build an analyzer for a real-world programming language is a complex engineering task. Chameleon abstracts all the technical details characterizing each language, allowing developers to focus on the analysis of dynamic features only and, hence, to define new analyses in a faster and simpler way. Furthermore, comparing similar control mechanisms, but built for different languages, is tricky. With Chameleon, is it possible to solve also this issue, since the analyses share the same underling language.

As a final remark, the interested reader can find the implementation of Chameleon at the following link: `https://github.com/samueleburo93/chameleon`.

References

[1] G. A. Kildall, "A unified approach to global program optimization," in Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ser. POPL '73, 1973, pp. 194–206.

[2] P. Cousot and R. Cousot, "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, ser. POPL '77, 1977, pp. 238–252.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," ACM Trans. Program. Lang. Syst., vol. 8, 1986, pp. 244–263.

[4] V. Arceri and S. Maffeis, "Abstract domains for type juggling," Electr. Notes Theor. Comput. Sci., vol. 331, 2017, pp. 41–55.

[5] S. Buro and I. Mastroeni, "Abstract code injection - A semantic approach based on abstract non-interference," in Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation, ser. VMCAI '18, 2018, pp. 116–137.

[6] I. Mastroeni and M. Pasqua, "Statically analyzing information flows: An abstract interpretation-based hyperanalysis for non-interference," in Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, ser. SAC '19, 2019, pp. 2215–2223.

[7] G. D. Plotkin, "A structural approach to operational semantics," Journal of Logic and Algebraic Programming, vol. 60-61, 2004, pp. 17–139. [Online]. Available: http://dx.doi.org/10.1016/j.jlap.2004.05.001

[8] M. Hennessy, The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[9] G. D. Plotkin, "A powerdomain for countable non-determinism," in Proceedings of the 9th International Colloquium on Automata, Languages and Programming, 1982, pp. 418–428.

 ISBN: 978-1-61208-755-9