Jinja: Towards a Comprehensive Formal Semantics for a Java-like Language

Tobias NIPKOW
Institut für Informatik
Technische Universität München

Abstract

Jinja is a Java-like programming language with a formal semantics designed to exhibit core features of Java. It is a compromise between realism of the language and tractability and clarity of the formal semantics. A big and a small step operational semantics are defined and shown equivalent. A type system and a definite initialization analysis are defined and type safety of the small step semantics is shown. The whole development has been carried out in the theorem prover Isabelle/HOL.

1 Introduction

There is a sizable body of literature on formal models of Java-like languages (e.g. [1, 2, 3, 8, 10, 15]). However, with the exception of [13], each model considers only particular aspects of the language. This extended abstract is a progress report on the way to a comprehensive model of a Java-like language (called **Jinja**) that encompasses both the source language and the virtual machine (we omit the latter aspect in this presentation). We are aiming for a language that is as realistic as possible but whose model is still comprehensible and where all proofs have been machine-checked. Although this work is partly one of unification and simplification, there are some new and noteworthy aspects: We present both a big and small step semantics and relate them (previously only one or the other was used), and we cover definite initialization of local variables. Last but not least all proofs are machine checked (in Isabelle/HOL [9]) but are still readable, although we do not have the space to go into this aspect [16, 7].

After a bit of basic notation (§2) and the introduction of Jinja (§3) we define a big step and a small step semantics (§4 and 5) and a type system (§6), and show type safety (§7), i.e. the reduction semantics respects the type system.

2 Basic Notation

In this section we introduce a few basic data types (of the meta language HOL) with their primitive operations. Note that $\Rightarrow$ is the space of total functions, that $'a$, $'b$, etc are type variables and that $t{::}\tau$ means that HOL term t has HOL type τ.

Pairs come with the two projection functions *fst* $:: {'a} \times {'b} \Rightarrow {'a}$ and *snd* $:: {'a} \times {'b} \Rightarrow {'b}$.

Lists (type $'a$ *list*) come with the empty list [], the infix constructor #, the infix @ that appends two lists, and the conversion function *set* from lists to sets. Variable names ending in "s" usually stand for lists. The datatype

datatype $'a$ *option* = *None* | *Some* $'a$

adjoins a new element *None* to a type $'a$.

Function update is written $f(x{:=}y)$ where $f{::}'a \Rightarrow {'b}$, $x{::}'a$ and $y{::}'b$. Given functions f and g of type $'a \Rightarrow {'b}$ and a set A of type $'a$ *set*, the function $f(g|A)$ is f overwritten with g restricted to A:

$f(g|A) \equiv \lambda a.$ ***if*** $a \in A$ ***then*** $g\ a$ ***else*** $f\ a$

Partial functions are modelled as functions of type $'a \Rightarrow {'b}$ *option*, where *None* represents undefinedness and $f\ x = Some\ y$ means x is mapped to y. We use the term **map** for such functions and abbreviate $f(x{:=}Some\ y)$ to $f(x{\mapsto}y)$. The latter notation extends to lists: $f([x_1,\ldots,x_n]\ [\mapsto]\ [y_1,\ldots,y_n])$ means $f(x_1{\mapsto}y_1)\ldots(x_n{\mapsto}y_n)$. We define $dom\ m \equiv \{a.\ m\ a \neq None\}$. The map $\lambda x.\ None$ is written *empty*.

Finally note that $[\![\ A_1; \ldots; A_n\ ]\!] \Longrightarrow A$ abbreviates the nested implication $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$. Occasionally we write "If A_1 and ... and A_n then A" too.

3 Jinja

Although Jinja is a typed language, we begin its description with the operational semantics which is independent of the type system. Hence we postpone the discussion of types until §6.1.

In the sequel we use the following variable conventions: V is a variable name, F a field name, M a method name, C a class name, e an expression, v a value, T a type, and P a program.

3.1 Values and Expressions

In Jinja, values can be primitive (*Bool* b where $b{::}bool$, and *Intg* i where $i{::}int$), or references (*Addr* a where a is an address), or the null reference *Null*, or the dummy value *Unit*. Jinja is an imperative but an expression-based language where statements are expressions that evaluate to *Unit*.

The following expressions are supported by Jinja: creation of new objects ($\mathit{New}\ C$), casting ($\mathit{Cast}\ C\ e$), values ($\mathit{Val}\ v$), variable access ($\mathit{Var}\ V$), binary operations ($e_1 \ll bop \gg e_2$ where bop is one of Add or Eq), variable assignment ($V{:=}e$), field access ($\{D\}e{\cdot}F$)[1], field assignment ($\{D\}e_1{\cdot}F{:=}e_2$), method call ($e{\cdot}M(es)$), block with locally declared variable ($\{V{:}T;\ e\}$), sequential composition ($e_1;\ e_2$), conditional ($\mathit{If}\ (e)\ e_1\ \mathit{Else}\ e_2$), loop ($\mathit{While}\ (e)\ e'$), exception throwing ($\mathit{Throw}\ e$) and catching ($\mathit{Try}\ e_1\ \mathit{Catch}(C\ V)\ e_2$). Note that there is no return statement because everything is an expression and returns a value. To ease notation we introduce some abbreviations:

$$\begin{array}{lcl}
\mathit{addr}\ a & \equiv & \mathit{Val}(\mathit{Addr}\ a)\\
\mathit{null} & \equiv & \mathit{Val}\ \mathit{Null}\\
\mathit{true} & \equiv & \mathit{Val}(\mathit{Bool}\ \mathit{True})\\
\mathit{false} & \equiv & \mathit{Val}(\mathit{Bool}\ \mathit{False})
\end{array}$$

Note that the annotation $\{D\}$ in field access and assignment is not part of the input language but is something that a preprocessor, e.g. the type checking phase of a compiler, must add. We come back to this point in §6.1.

3.2 Declarations

Everything — expression evaluation, type checking, etc — is performed in the context of a program P. We do not go into the structure of programs here but work in terms of a few abstract constructs for analyzing and accessing the declarations in a program:

is-class P C means class C is defined in P.

$\boldsymbol{P \vdash D \preceq_C C}$ means D is a **subclass** of C. It is transitive and reflexive.

$\boldsymbol{P \vdash C}$ ***sees-method*** $\boldsymbol{M{:}Ts{\to}T = (pns,body)}$ ***in D*** means that in P from class C a method M is visible in class D (taking overriding into account) with argument types Ts (a type list), result type T, formal parameter list pns, and body $body$.

$\boldsymbol{P \vdash C}$ ***sees-field*** $\boldsymbol{F{:}T}$ ***in D*** means that in P from class C a field F of type T is visible in class D.

$\boldsymbol{P \vdash C}$ ***has-field*** $\boldsymbol{F{:}T}$ ***in D*** means that in P a (not necessarily proper) superclass D of C has a field F of type T.

The following example (in an imaginary syntax) should clarify the concepts:

```
class B extends A {field F:TB
                   method M:TBs->T1 = (pB,bB)}
class C extends B {field F:TC
                   method M:TCs->T2 = (pC,bC)}
```

[1] D is the class where F is declared; we write $e{\cdot}F$ because the usual $e.F$ would clash with Isabelle's lexical syntax for qualified names.

We have $P \vdash C$ *sees-field* $F{:}TC$ *in* C but not $P \vdash C$ *sees-field* $F{:}TB$ *in* B because the declaration in C hides the one in B. In contrast, we have both $P \vdash C$ *has-field* $F{:}TC$ *in* C and $P \vdash C$ *has-field* $F{:}TB$ *in* B because *has-field* is independent of visibility.

Analogously we have $P \vdash B$ *sees-method* M: $TBs \rightarrow T_1 = (pB,\ bB)$ *in* B and $P \vdash C$ *sees-method* M: $TCs \rightarrow T_2 = (pC,\ bC)$ *in* C, but not $P \vdash C$ *sees-method* M: $TBs \rightarrow T_1 = (pB,\ bB)$ *in* B. The second declaration of M overrides the first, no matter what the type of M in the two declarations is. This differs from Java, where methods can also be "overloaded", which means that multiple declarations of M can be visible simultaneously, provided they are distinguished by their argument types. We have formalized overloading elsewhere [11] but have not included it in Jinja: it complicates matters without adding a significant new aspect, and it can make programs hard to understand.

4 Big Step Semantics

4.1 State

The **state** during expression evaluation is a pair of a **heap** and a **store**. A store is a map from variable names to values. A heap is map from addresses to objects. An object is a pair of a class name and a field table, and a **field table** is a map from pairs $(F,\ D)$ (where D is the class where F is declared) to values. It is essential to include D because an object may have multiple fields of the same name. The variable convention is that h is a heap, l is a store (the *l*ocal variables), and s a state. The projection functions *hp* and *lcl* are synonyms for *fst* and *snd*.

4.2 Evaluation

The evaluation judgement is of the form $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$, where e and s are the initial expression and state, and e' and s' the final expression and state. We then say that e **evaluates to** e'. The rules will be such that final expressions are always values (*Val*) or exceptions (*Throw*), i.e. final expressions are completely evaluated. The full set of evaluation rules is shown in Appendix A grouped by construct. We will now discuss them in an incremental fashion: first normal evaluation only, exceptional behaviour afterwards.

4.2.1 Normal Evaluation

Normal evaluation means that we are defining an exception-free language. In particular, all final expressions will be values. We start with the evaluation of basic expressions:

$$\llbracket \textit{new-Addr}\ h = \textit{Some}\ a;\ P \vdash C\ \textit{has-fields}\ \textit{FDTs};\\ \quad h' = h(a \mapsto (C,\ \textit{init-vars}\ \textit{FDTs}))\rrbracket\\ \Longrightarrow P \vdash \langle \textit{New}\ C,(h,\ l)\rangle \Rightarrow \langle \textit{addr}\ a,(h',\ l)\rangle$$

$$\llbracket P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{addr}\ a,(h,\ l)\rangle;\ h\ a = \textit{Some}\ (D,\ \textit{fs});\ P \vdash D \preceq_C C\rrbracket\\ \Longrightarrow P \vdash \langle \textit{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{addr}\ a,(h,\ l)\rangle$$

$$P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{null},s_1\rangle \Longrightarrow P \vdash \langle \textit{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{null},s_1\rangle$$

$$P \vdash \langle \textit{Val}\ v,s\rangle \Rightarrow \langle \textit{Val}\ v,s\rangle$$

$$l\ V = \textit{Some}\ v \Longrightarrow P \vdash \langle \textit{Var}\ V,(h,\ l)\rangle \Rightarrow \langle \textit{Val}\ v,(h,\ l)\rangle$$

$$\llbracket P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{Val}\ v,(h,\ l)\rangle;\ l' = l(V \mapsto v)\rrbracket\\ \Longrightarrow P \vdash \langle V{:=}e,s_0\rangle \Rightarrow \langle \textit{Val}\ v,(h,\ l')\rangle$$

$$\llbracket P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{addr}\ a,(h,\ l)\rangle;\ h\ a = \textit{Some}\ (C,\ \textit{fs});\ \textit{fs}\ (F,\ D) = \textit{Some}\ v\rrbracket\\ \Longrightarrow P \vdash \langle \{D\}e{\cdot}F,s_0\rangle \Rightarrow \langle \textit{Val}\ v,(h,\ l)\rangle$$

$$\llbracket P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \textit{addr}\ a,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \textit{Val}\ v,(h_2,\ l_2)\rangle;\\ \quad h_2\ a = \textit{Some}\ (C,\ \textit{fs});\ \textit{fs}' = \textit{fs}((F,\ D) \mapsto v);\ h_2' = h_2(a \mapsto (C,\ \textit{fs}'))\rrbracket\\ \Longrightarrow P \vdash \langle \{D\}e_1{\cdot}F{:=}e_2,s_0\rangle \Rightarrow \langle \textit{Val}\ v,(h_2',\ l_2)\rangle$$

$$\llbracket P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \textit{Val}\ v_1,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \textit{Val}\ v_2,s_2\rangle\rrbracket\\ \Longrightarrow P \vdash \langle e_1 \ll \textit{bop} \gg e_2,s_0\rangle \Rightarrow \langle \textit{Val}\ (\textit{binop}\ \textit{bop}\ v_1\ v_2),s_2\rangle$$

New C allocates a new address and initializes it with an object where all fields are set to their default values: function *new-Addr* returns a "new" address, i.e. *new-Addr h* = *Some a* implies *h a* = *None*; relation *has-fields* computes the list *FDTs* of all field declarations in and above C, where each field F of type T declared in class D is represented as a triple $((F,D),T)$; and *init-vars FDTs* maps each pair (F,D) to the default value of type T — the definition of the default value is irrelevant for our purposes, it suffices to know that it is *Some* rather than *None*.

There are two rules for *Cast C e*: if e evaluates to the address of an object of a subclass of C or to *null*, the cast succeeds, in the latter case because the null reference is in every class.

Field access $\{D\}e{\cdot}F$ evaluates e to an address, looks up the object at the address, indexes its field table with (F,D), and evaluates to the value found in the field table. Note that field lookup follows a static binding discipline: the dynamic class C is ignored and the annotation D is used instead. Later on well-typedness will require D to be the first class where F is declared when we start looking from the static class of e up the class hierarchy.

Field assignment $\{D\}e_1{\cdot}F{:=}e_2$ evaluates e_1 to an address and e_2 to a value, updates the object at the address with the value (using the index (F,D)), and evaluates to that value.

Binary operations are evaluated from left to right. Function *binop* takes a binary operation and two values and applies the operation to the values — its precise definition is not important here.

Next we consider the evaluation rules for blocks:

$P \vdash \langle e_0,(h_0,\ l_0(V := \mathit{None}))\rangle \Rightarrow \langle e_1,(h_1,\ l_1)\rangle \Longrightarrow$
$P \vdash \langle\{V{:}T;\ e_0\},(h_0,\ l_0)\rangle \Rightarrow \langle e_1,(h_1,\ l_1(V := l_0\ V))\rangle$

In a block, the expression is evaluated in the context of a store where the local variable has been removed, i.e. set to *None*. Afterwards the original value of the variable in the initial store is restored.

The lengthiest rule is the one for method call:

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{addr}\ a,s_1\rangle;$
$\quad P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle \mathit{map\ Val\ vs},(\lambda u.\ \mathit{Some}\ (\mathit{C\text{-}3}\ u,\ \mathit{fs\text{-}2}\ u),\ l_2)\rangle;$
$\quad P \vdash \mathit{C\text{-}3}\ a\ \mathit{sees\text{-}method}\ M{:}\ Ts{\rightarrow}T = (pns,\ body)\ \mathit{in}\ D;\ \mathit{length}\ vs = \mathit{length}\ Ts;$
$\quad l_2{}' = l_2(\mathit{this} \mapsto \mathit{Addr}\ a,\ pns\ [\mapsto]\ vs);$
$\quad P \vdash \langle body,(\lambda u.\ \mathit{Some}\ (\mathit{C\text{-}3}\ u,\ \mathit{fs\text{-}2}\ u),\ l_2{}')\rangle \Rightarrow \langle e',(h_3,\ l_3)\rangle;$
$\quad l_3{}' = l_3(l_2|\{\mathit{this}\} \cup \mathit{set}\ pns)]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle e',(h_3,\ l_3{}')\rangle$

Its reading is easy: evaluate e to an address a and the parameter list ps to a list of values vs ($[\Rightarrow]$ is evaluation extended to lists of expressions), look up the class C of the object in the heap at a, look up the parameter names *pns* and body *body* of the method M visible from C, extend the store by mapping the *this* pointer to *Addr a* and the formal parameter names to the actual parameter values, evaluate the body in this extended store, and finally reset the overwritten store variables *this* and *pns* to their original values.

What may puzzle is that we evaluate the body in the context of all of $l_2(\mathit{this}\ \#\ pns\ [\mapsto]\ \mathit{Addr}\ a\ \#\ vs)$ rather than just $\mathit{empty}(\mathit{this}\ \#\ pns\ [\mapsto]\ \mathit{Addr}\ a\ \#\ vs)$, which would also obviate the need to reset the overwritten variables at the end. That is indeed a perfectly reasonable semantics, but one that builds in that the body has no access to variables other than *this* and the parameters. We prefer to leave the operational semantics as general as possible. As a consequence, we obtain dynamic variable binding: any non-local variable V in the *body* will refer to the most recently created instance of V in the store. If this language feature is deemed undesirable (as it generally is today), one can rule such programs out by means of the type system, which we will do later on.

In Jinja, sequential composition, conditional and while-loop are expressions too, in contrast to Java, where they are commands and do not return a value. Their evaluation rules are straightforward:

$[\![P \vdash \langle e_0,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash \langle e_1,s_1\rangle \Rightarrow \langle e_2,s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e_0;\ e_1,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{true},s_1\rangle;\ P \vdash \langle e_1,s_1\rangle \Rightarrow \langle e',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{If}\ (e)\ e_1\ \mathit{Else}\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{false},s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle e',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{If}\ (e)\ e_1\ \mathit{Else}\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{false},s_1\rangle \Longrightarrow P \vdash \langle \mathit{While}\ (e)\ c,s_0\rangle \Rightarrow \langle \mathit{Val\ Unit},s_1\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{true},s_1\rangle;\ P \vdash \langle c,s_1\rangle \Rightarrow \langle \mathit{Val}\ v_1,s_2\rangle;$
$\quad P \vdash \langle \mathit{While}\ (e)\ c,s_2\rangle \Rightarrow \langle e_3,s_3\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{While}\ (e)\ c,s_0\rangle \Rightarrow \langle e_3,s_3\rangle$

Sequential composition discards the value of the first expression. Similarly, while-loops discard the value of their body and, upon termination, return *Unit*.

It only remains to define [⇒], the evaluation of expression lists, needed for method calls. The rules express that lists are evaluated from left to right:

$P \vdash \langle [],s\rangle\ [\Rightarrow]\ \langle [],s\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e\ \#\ es,s_0\rangle\ [\Rightarrow]\ \langle \mathit{Val}\ v\ \#\ es',s_2\rangle$

We have now seen the complete semantics of an exception-free fragment of Jinja.

4.3 Exceptions

The rules above assume that during evaluation everything fits together. If it does not, the semantics gets stuck, i.e. there is no final value. For example, evaluation of $\langle \mathit{Var}\ V,\ (h,l)\rangle$ only succeeds if $V \in \mathit{dom}\ l$. Later on, a static analysis ("definite initialization") will identify expressions where $V \in \mathit{dom}\ l$ always holds. Thus we do not need a rule for the situation where $V \notin \mathit{dom}\ l$. In contrast, many exceptional situations arise because of null references which we deal with by raising an exception. That is, the expression does not evaluate to a normal value but to an exception $\mathit{Throw}(\mathit{addr}\ a)$ where a is the address of some object, the exception object.

There are both system and user exceptions. User exceptions can refer to arbitrary objects. System exceptions refer to an object in one of three system exception classes *NullPointer*, *ClassCast* and *OutOfMemory* — their names speak for themselves. Since system exception objects do not carry any information in addition to their class name, we can simplify their treatment by pre-allocating one object for each system exception class. Thus a few addresses are reserved for pre-allocated system exception objects. This is modelled by a function *addr-of-sys-xcpt* from class names to addresses whose precise definition is not important here. To ease notation we introduce some abbreviations:

$$\begin{array}{lcl} \mathit{THROW}\ a & \equiv & \mathit{Throw}(\mathit{addr}\ a) \\ \mathit{throw}\ C & \equiv & \mathit{THROW}(\mathit{addr\text{-}of\text{-}sys\text{-}xcpt}\ C) \end{array}$$

4.4 Exceptional Evaluation

In the following situations system exceptions are thrown: if there is no more free storage, if a cast fails, or if the object reference in a field access or update or a method call is null:

$new\text{-}Addr\ h = None \Longrightarrow P \vdash \langle New\ C,(h,\ l)\rangle \Rightarrow \langle throw\ OutOfMemory,(h,\ l)\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,(h,\ l)\rangle;\ h\ a = Some\ (D,\ fs);\ \neg\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle throw\ ClassCast,(h,\ l)\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow P \vdash \langle \{D\}e{\cdot}F,s_0\rangle \Rightarrow \langle throw\ NullPointer,s_1\rangle$

$[\![P \vdash \langle e_1,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle Val\ v,s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle \{D\}e_1{\cdot}F{:=}e_2,s_0\rangle \Rightarrow \langle throw\ NullPointer,s_2\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle;\ P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle map\ Val\ vs,s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle throw\ NullPointer,s_2\rangle$

Note that we have maintained Java's eager evaluation scheme of evaluating all subterms before throwing any system exception.

Exceptions can also be thrown explicitly — any expression of class type can be thrown. However, throwing *null* raises the *NullPointer* exception.

$P \vdash \langle e,s_0\rangle \Rightarrow \langle addr\ a,s_1\rangle \Longrightarrow P \vdash \langle Throw\ e,s_0\rangle \Rightarrow \langle THROW\ a,s_1\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle null,s_1\rangle \Longrightarrow P \vdash \langle Throw\ e,s_0\rangle \Rightarrow \langle throw\ NullPointer,s_1\rangle$

Thrown exceptions can be caught using the construct $Try\ e_1\ Catch(C\ V)\ e_2$. If e_1 evaluates to a value, the whole expression evaluates to that value. If e_1 evaluates to an exception $THROW\ a$ such that a refers to an object of a subclass of C, V is set to $Addr\ a$ and e_2 is evaluated; otherwise $THROW\ a$ is the result of the evaluation.

$P \vdash \langle e_1,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle \Longrightarrow P \vdash \langle Try\ e_1\ Catch(C\ V)\ e_2,s_0\rangle \Rightarrow \langle Val\ v_1,s_1\rangle$

$[\![P \vdash \langle e_1,s_0\rangle \Rightarrow \langle THROW\ a,(h_1,\ l_1)\rangle;\ h_1\ a = Some\ (D,\ fs);\ P \vdash D \preceq_C C;$
$\quad P \vdash \langle e_2,(h_1,\ l_1(V \mapsto Addr\ a))\rangle \Rightarrow \langle e_2{}',(h_2,\ l_2)\rangle]\!]$
$\Longrightarrow P \vdash \langle Try\ e_1\ Catch(C\ V)\ e_2,s_0\rangle \Rightarrow \langle e_2{}',(h_2,\ l_2(V := l_1\ V))\rangle$

$[\![P \vdash \langle e_1,s_0\rangle \Rightarrow \langle THROW\ a,(h_1,\ l_1)\rangle;\ h_1\ a = Some\ (D,\ fs);\ \neg\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle Try\ e_1\ Catch(C\ V)\ e_2,s_0\rangle \Rightarrow \langle THROW\ a,(h_1,\ l_1)\rangle$

Finally, exceptions must be propagated. That is, if the evaluation of a certain subexpression throws an exception, the evaluation of the whole expression has to throw that exception. The exception propagation rules are straightforward:

$P \vdash \langle e,s_0\rangle \Rightarrow \langle Throw\ e',s_1\rangle \Longrightarrow P \vdash \langle Cast\ C\ e,s_0\rangle \Rightarrow \langle Throw\ e',s_1\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle V{:=}e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle \{D\}e{\cdot}F,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle \{D\}e_1{\cdot}F{:=}e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$[\![P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Throw}\ e',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle \{D\}e_1{\cdot}F{:=}e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_2\rangle$

$P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_1\rangle \Longrightarrow P \vdash \langle e_1 \ll\!\mathit{bop}\!\gg e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_1\rangle$

$[\![P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Val}\ v_1,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Throw}\ e,s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e_1 \ll\!\mathit{bop}\!\gg e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_2\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle;$
$\quad es' = \mathit{map}\ \mathit{Val}\ vs\ @\ \mathit{Throw}\ ex\ \#\ es_2]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle \mathit{Throw}\ ex,s_2\rangle$

$P \vdash \langle e_0,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_1\rangle \Longrightarrow P \vdash \langle e_0;\ e_1,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_1\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow$
$P \vdash \langle \mathit{If}\ (e)\ e_1\ \mathit{Else}\ e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle \mathit{While}\ (e)\ c,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{true},s_1\rangle;\ P \vdash \langle c,s_1\rangle \Rightarrow \langle \mathit{Throw}\ e',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{While}\ (e)\ c,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_2\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle \mathit{Throw}\ e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle e\ \#\ es,s_0\rangle\ [\Rightarrow]\ \langle \mathit{Throw}\ e'\ \#\ es,s_1\rangle$

This concludes the exposition of the evaluation rules.

A compact representation of the above exception propagation rules can be achieved by introducing the notion of a *context* C_x (essentially a grammar for positions in expressions where exceptions propagate to the top) and by giving one rule $\langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow \langle C_x[e],s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$. We prefer not to formalize these additional notions and stay within a fixed basic framework of ordinary expressions.

4.5 Final Expressions

Now that we have the complete set of rules we can show that evaluation always produces a **final** expression:

Definition $\mathit{final}\ e \equiv (\exists\, v.\ e = \mathit{Val}\ v) \lor (\exists\, a.\ e = \mathit{THROW}\ a)$

Lemma 4.1 If $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ then *final* e'.

The proof is by induction on the evaluation relation $\Rightarrow$. Since the latter is defined simultaneously with the evaluation relation $[\Rightarrow]$ for expression lists, we need to prove a proposition about $[\Rightarrow]$ simultaneously with Lemma 4.1. This will also be the common proof pattern in all other inductive proofs about $\Rightarrow$. In most cases the statement about $[\Rightarrow]$ is a lifted version of the one about $[\Rightarrow]$. In the above case one might expect something like $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle \Longrightarrow \forall\, e'{\in}set\ es'.\ final\ e'$. However, this is wrong: due to exceptions, evaluation may stop before the end of the list. A final expression list is a list of values, possibly followed by a *THROW* and some further expressions:

Definition *finals es* $\equiv$
$(\exists\, vs.\ es = map\ Val\ vs) \vee (\exists\, vs\ a\ es'.\ es = map\ Val\ vs\ @\ THROW\ a\ \#\ es')$

The version of Lemma 4.1 for lists is now simply

If $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es',s'\rangle$ then *finals* es'.

It is equally straightforward to prove that final expressions evaluate to themselves:

Lemma 4.2 If *final* e then $P \vdash \langle e,s\rangle \Rightarrow \langle e,s\rangle$. If *finals es* then $P \vdash \langle es,s\rangle\ [\Rightarrow]\ \langle es,s\rangle$.

5 Small Step Semantics

Because of its simplicity, a big step semantics has several drawbacks. For example, it cannot accommodate parallelism, a potentially desirable extension of Jinja. The reason is that $\Rightarrow$ cannot talk about the intermediate states during evaluation. For the same reason the type safety proof in §7 needs a finer grained semantics. Thus we now move over to an equivalent small step semantics.

The judgement for the small step semantics is $P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle$ and describes a single micro-step in the evaluation of e towards its final value. We say that e **reduces to** e' (in one step). Below we will compose sequences of such single steps $\langle e_1,s_1\rangle \rightarrow \langle e_2,s_2\rangle \ldots \rightarrow \langle e_n,s_n\rangle$ to reduce an expression completely.

As for the big step semantics we can define normal and exceptional reductions separately. The complete set of reduction rules is shown in Appendix B grouped by construct.

5.1 Normal Reduction

The reduction rules come in two flavours: those that reduce a subexpression of an expression and those that reduce the whole expression. The former have no counterpart in the big step semantics as they are handled implicitly in the premises of the big step rules.

5.1.1 Subexpression Reduction

These rules essentially describe in which order subexpressions are evaluated. Therefore most of them follow a common pattern:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle c\ \ldots\ e\ \ldots,\ s\rangle \rightarrow \langle c\ \ldots\ e'\ \ldots,\ s'\rangle$

where c is a constructor and e and e' are meta-variables. The other subexpressions of c may be more complex to indicate, for example, which of them must be values already, thus expressing the order of reduction. The rules for basic expressions

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle \mathit{Cast}\ C\ e,s\rangle \rightarrow \langle \mathit{Cast}\ C\ e',s'\rangle$

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle V{:=}e,s\rangle \rightarrow \langle V{:=}e',s'\rangle$

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle\{D\}e{\cdot}F,s\rangle \rightarrow \langle\{D\}e'{\cdot}F,s'\rangle$

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle\{D\}e{\cdot}F{:=}e_2,s\rangle \rightarrow \langle\{D\}e'{\cdot}F{:=}e_2,s'\rangle$

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle\{D\}\mathit{Val}\ v{\cdot}F{:=}e,s\rangle \rightarrow \langle\{D\}\mathit{Val}\ v{\cdot}F{:=}e',s'\rangle$

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e \ll\!\mathit{bop}\!\gg e_2,s\rangle \rightarrow \langle e' \ll\!\mathit{bop}\!\gg e_2,s'\rangle$

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle \mathit{Val}\ v_1 \ll\!\mathit{bop}\!\gg e,s\rangle \rightarrow \langle \mathit{Val}\ v_1 \ll\!\mathit{bop}\!\gg e',s'\rangle$

follow this pattern exactly. For example, the rules for field assignment express that the left-hand side is evaluated before the right-hand side.

The rules for blocks are more complicated:

$[\![P \vdash \langle e,(h,\ l(V := \mathit{None}))\rangle \rightarrow \langle e',(h',\ l')\rangle;\ l'\ V = \mathit{None};\ \neg\ \mathit{assigned}\ V\ e]\!]$
$\Longrightarrow P \vdash \langle\{V{:}T;\ e\},(h,\ l)\rangle \rightarrow \langle\{V{:}T;\ e'\},(h',\ l'(V := l\ V))\rangle$

$[\![P \vdash \langle e,(h,\ l(V := \mathit{None}))\rangle \rightarrow \langle e',(h',\ l')\rangle;\ l'\ V = \mathit{Some}\ v;\ \neg\ \mathit{assigned}\ V\ e]\!]$
$\Longrightarrow P \vdash \langle\{V{:}T;\ e\},(h,\ l)\rangle \rightarrow \langle\{V{:}T;\ V{:=}\mathit{Val}\ v;\ e'\},(h',\ l'(V := l\ V))\rangle$

$[\![P \vdash \langle e,(h,\ l(V \mapsto v))\rangle \rightarrow \langle e',(h',\ l')\rangle;\ l'\ V = \mathit{Some}\ v']\!]$
$\Longrightarrow P \vdash \langle\{V{:}T;\ V{:=}\mathit{Val}\ v;\ e\},(h,\ l)\rangle \rightarrow$
$\langle\{V{:}T;\ V{:=}\mathit{Val}\ v';\ e'\},(h',\ l'(V := l\ V))\rangle$

In a block $\{V{:}T;\ e\}$ we keep reducing e in a store where V is undefined (*None*), restoring the original binding of V after each step. Once the store after the reduction step binds V to a value v, this binding is remembered by adding an assignment in front of the reduced expression, yielding $\{V{:}T;\ V{:=}\,\mathit{Val}\ v;\ e'\}$. The final rule reduces such blocks. This additional rule is necessary because $\{V{:}T;\ V{:=}\,\mathit{Val}\ v;\ e\}$ must not be reduced by reducing all of $V{:=}\,\mathit{Val}\ v;\ e$, which would merely reduce $V{:=}\,\mathit{Val}\ v$, but by reducing e. To avoid these undesirable reductions we have introduced the predicate

Definition $\mathit{assigned}\ V\ e \equiv \exists\, v\ e'.\ e = V{:=}\,\mathit{Val}\ v;\ e'$

and added it as a precondition to the initial two reduction rules.

Note that we cannot treat local variables simply by creating "new" variables because because we do not know which other variables exist in the context: *dom l* does not contain all of them because variables need not be initialized upon creation, something that other semantics often assume.

Sequential composition and conditional are self explanatory:

$$P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow P \vdash \langle e;\ e_2,s\rangle \to \langle e';\ e_2,s'\rangle$$

$$P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow P \vdash \langle \mathit{If}\ (e)\ e_1\ \mathit{Else}\ e_2,s\rangle \to \langle \mathit{If}\ (e')\ e_1\ \mathit{Else}\ e_2,s'\rangle$$

To reduce a method call, the object expression is reduced until it has become an address, and then the parameters are reduced:

$$P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow P \vdash \langle e{\cdot}M(es),s\rangle \to \langle e'{\cdot}M(es),s'\rangle$$

$$P \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle \Longrightarrow P \vdash \langle \mathit{Val}\ v{\cdot}M(es),s\rangle \to \langle \mathit{Val}\ v{\cdot}M(es'),s'\rangle$$

The relation $[\to]$ is the extension of $\to$ to expression lists. Both relations are defined simultaneously. Lists are reduced from left to right, each element is reduced until it has become a value:

$$P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow P \vdash \langle e\ \#\ es,s\rangle\ [\to]\ \langle e'\ \#\ es,s'\rangle$$

$$P \vdash \langle es,s\rangle\ [\to]\ \langle es',s'\rangle \Longrightarrow P \vdash \langle \mathit{Val}\ v\ \#\ es,s\rangle\ [\to]\ \langle \mathit{Val}\ v\ \#\ es',s'\rangle$$

5.1.2 Expression Reduction

Once the subexpressions are sufficiently reduced, we can reduce the whole expression. The rules for basic expressions are fairly obvious

$$[\![\mathit{new\text{-}Addr}\ h = \mathit{Some}\ a;\ P \vdash C\ \mathit{has\text{-}fields}\ \mathit{FDTs}]\!]$$
$$\Longrightarrow P \vdash \langle \mathit{New}\ C,(h,\ l)\rangle \to \langle \mathit{addr}\ a,(h(a \mapsto (C,\ \mathit{init\text{-}vars}\ \mathit{FDTs})),\ l)\rangle$$

$$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (D,\ \mathit{fs});\ P \vdash D \preceq_C C]\!]$$
$$\Longrightarrow P \vdash \langle \mathit{Cast}\ C\ (\mathit{addr}\ a),s\rangle \to \langle \mathit{addr}\ a,s\rangle$$

$P \vdash \langle \mathit{Cast}\ C\ \mathit{null},s\rangle \to \langle \mathit{null},s\rangle$

$\mathit{lcl}\ s\ V = \mathit{Some}\ v \Longrightarrow P \vdash \langle \mathit{Var}\ V,s\rangle \to \langle \mathit{Val}\ v,s\rangle$

$P \vdash \langle V{:=}\mathit{Val}\ v,(h,\ l)\rangle \to \langle \mathit{Val}\ v,(h,\ l(V \mapsto v))\rangle$

$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (C,\ \mathit{fs});\ \mathit{fs}\ (F,\ D) = \mathit{Some}\ v]\!]$
$\Longrightarrow P \vdash \langle \{D\}\mathit{addr}\ a{\cdot}F,s\rangle \to \langle \mathit{Val}\ v,s\rangle$

$h\ a = \mathit{Some}\ (C,\ \mathit{fs}) \Longrightarrow$
$P \vdash \langle \{D\}\mathit{addr}\ a{\cdot}F{:=}\mathit{Val}\ v,(h,\ l)\rangle \to \langle \mathit{Val}\ v,(h(a \mapsto (C,\ \mathit{fs}((F,\ D) \mapsto v))),\ l)\rangle$

$P \vdash \langle \mathit{Val}\ v_1 \ll\!\mathit{bop}\!\gg \mathit{Val}\ v_2,s\rangle \to \langle \mathit{Val}\ (\mathit{binop}\ \mathit{bop}\ v_1\ v_2),s\rangle$

and resemble their big step counterparts. Reduction of blocks is equally clear:

$P \vdash \langle \{V{:}T;\ V{:=}\mathit{Val}\ v;\ \mathit{Val}\ u\},s\rangle \to \langle \mathit{Val}\ u,s\rangle$

$P \vdash \langle \{V{:}T;\ \mathit{Val}\ u\},s\rangle \to \langle \mathit{Val}\ u,s\rangle$

The rule for method invocation is pleasingly simple:

$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (C,\ \mathit{fs});\ P \vdash C\ \textit{sees-method}\ M{:}\ \mathit{Ts}{\to}T = (\mathit{pns},\ \mathit{body})\ \mathit{in}\ D;$
$\quad \mathit{length}\ \mathit{vs} = \mathit{length}\ \mathit{Ts}]\!]$
$\Longrightarrow P \vdash \langle \mathit{addr}\ a{\cdot}M(\mathit{map}\ \mathit{Val}\ \mathit{vs}),s\rangle \to$
$\qquad \langle \mathit{blocks}\ (\mathit{this}\ \#\ \mathit{pns},\ \mathit{Class}\ D\ \#\ \mathit{Ts},\ \mathit{Addr}\ a\ \#\ \mathit{vs},\ \mathit{body}),s\rangle$

In order to avoid explicit stacks we use blocks to hold the values of the parameters. The required nested block structure is built with the help of the auxiliary function *blocks* of type *vname list* × *ty list* × *val list* × *expr* ⇒ *expr*. In functional programming style:

$\mathit{blocks}\ (V\ \#\ \mathit{Vs},\ T\ \#\ \mathit{Ts},\ v\ \#\ \mathit{vs},\ e) = \{V{:}T;\ V{:=}\mathit{Val}\ v;\ \mathit{blocks}\ (\mathit{Vs},\ \mathit{Ts},\ \mathit{vs},\ e)\}$
$\mathit{blocks}\ (\mathit{Vs},\ \mathit{Ts},\ \mathit{vs},\ e) = e$

The rules for sequential composition, conditional and while-loop are again as expected — the one for while-loops is particularly economic:

$P \vdash \langle \mathit{Val}\ v;\ e_2,s\rangle \to \langle e_2,s\rangle$

$P \vdash \langle \mathit{If}\ (\mathit{true})\ e_1\ \mathit{Else}\ e_2,s\rangle \to \langle e_1,s\rangle$

$P \vdash \langle \mathit{If}\ (\mathit{false})\ e_1\ \mathit{Else}\ e_2,s\rangle \to \langle e_2,s\rangle$

$P \vdash \langle \mathit{While}\ (b)\ c,s\rangle \to \langle \mathit{If}\ (b)\ (c;\ \mathit{While}\ (b)\ c)\ \mathit{Else}\ \mathit{Val}\ \mathit{Unit},s\rangle$

5.2 Exceptional Reduction

We begin with the rules for throwing system exceptions which resemble those for the big step semantics closely:

$\mathit{new\text{-}Addr}\ h = \mathit{None} \Longrightarrow P \vdash \langle \mathit{New}\ C,(h,\ l)\rangle \rightarrow \langle \mathit{throw}\ \mathit{OutOfMemory},(h,\ l)\rangle$

$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (D,\ \mathit{fs});\ \neg\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle \mathit{Cast}\ C\ (\mathit{addr}\ a),s\rangle \rightarrow \langle \mathit{throw}\ \mathit{ClassCast},s\rangle$

$P \vdash \langle \{T\}\mathit{null}\cdot F,s\rangle \rightarrow \langle \mathit{throw}\ \mathit{NullPointer},s\rangle$

$P \vdash \langle \{D\}\mathit{null}\cdot F{:=}\mathit{Val}\ v,s\rangle \rightarrow \langle \mathit{throw}\ \mathit{NullPointer},s\rangle$

$P \vdash \langle \mathit{null}\cdot M(\mathit{map}\ \mathit{Val}\ \mathit{vs}),s\rangle \rightarrow \langle \mathit{throw}\ \mathit{NullPointer},s\rangle$

We can reduce underneath a *Throw* and reduce it to a *NullPointer* exception if necessary:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle \mathit{Throw}\ e,s\rangle \rightarrow \langle \mathit{Throw}\ e',s'\rangle$

$P \vdash \langle \mathit{Throw}\ \mathit{null},s\rangle \rightarrow \langle \mathit{throw}\ \mathit{NullPointer},s\rangle$

This is how $\mathit{Try}\ e\ \mathit{Catch}(C\ V)\ e_2$ is reduced:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow$
$P \vdash \langle \mathit{Try}\ e\ \mathit{Catch}(C\ V)\ e_2,s\rangle \rightarrow \langle \mathit{Try}\ e'\ \mathit{Catch}(C\ V)\ e_2,s'\rangle$

$P \vdash \langle \mathit{Try}\ \mathit{Val}\ v\ \mathit{Catch}(C\ V)\ e_2,s\rangle \rightarrow \langle \mathit{Val}\ v,s\rangle$

$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (D,\ \mathit{fs});\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle \mathit{Try}\ \mathit{THROW}\ a\ \mathit{Catch}(C\ V)\ e_2,s\rangle \rightarrow \langle \{V{:}\mathit{Class}\ C;\ V{:=}\mathit{addr}\ a;\ e_2\},s\rangle$

$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (D,\ \mathit{fs});\ \neg\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle \mathit{Try}\ \mathit{THROW}\ a\ \mathit{Catch}(C\ V)\ e_2,s\rangle \rightarrow \langle \mathit{THROW}\ a,s\rangle$

First we must reduce e_1. If it becomes a value, the whole expression evaluates to that value. If it becomes a *THROW a*, there are two possibilities: if a can be caught, the term reduces to a block with V set to a and body e_2, otherwise the exception is propagated.

Exception propagation for all other constructs is straightforward:

$P \vdash \langle \mathit{Cast}\ C\ (\mathit{Throw}\ e),s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle V{:=}\mathit{Throw}\ e,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \{T\}\mathit{Throw}\ e\cdot F,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \{D\}\mathit{Throw}\ e\cdot F{:=}e_2,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \{D\}\mathit{Val}\ v\cdot F{:=}\mathit{Throw}\ e,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \mathit{Throw}\ e \ll\!\mathit{bop}\!\gg e_2,s\rangle \to \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \mathit{Val}\ v_1 \ll\!\mathit{bop}\!\gg \mathit{Throw}\ e,s\rangle \to \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle\{V{:}T;\ \mathit{THROW}\ a\},s\rangle \to \langle \mathit{THROW}\ a,s\rangle$

$P \vdash \langle\{V{:}T;\ V{:=}\mathit{Val}\ v;\ \mathit{THROW}\ a\},s\rangle \to \langle \mathit{THROW}\ a,s\rangle$

$P \vdash \langle \mathit{Throw}\ e{\cdot}M(\mathit{es}),s\rangle \to \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \mathit{Val}\ v{\cdot}M(\mathit{map}\ \mathit{Val}\ \mathit{vs}\ @\ \mathit{Throw}\ e\ \#\ \mathit{es}'),s\rangle \to \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \mathit{Throw}\ e;\ e_2,s\rangle \to \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \mathit{If}\ (\mathit{Throw}\ e)\ e_1\ \mathit{Else}\ e_2,s\rangle \to \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \mathit{Throw}\ (\mathit{Throw}\ e),s\rangle \to \langle \mathit{Throw}\ e,s\rangle$

It should be noted that $\{V{:}T;\ \mathit{Throw}\ e\}$ can in general not be reduced to *Throw* e because e may refer to the local V which must not escape its scope. Hence e must be reduced to an address first.

5.2.1 The Reflexive Transitive Closure

If we write $P \vdash \langle e_1,s_1\rangle \to^* \langle e_n,s_n\rangle$ this means that there is a sequence of reductions $P \vdash \langle e_1,s_1\rangle \to \langle e_2,s_2\rangle$, $P \vdash \langle e_2,s_2\rangle \to \langle e_3,s_3\rangle$ ..., and similarly for $[\to]$ and $[\to]^*$

5.3 Relating Big Step and Small Step Semantics

Our big and small step semantics are equivalent in the following sense:

Theorem 5.1 If *wf-J-prog* P then
$P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ iff $P \vdash \langle e,s\rangle \to^* \langle e',s'\rangle \land \mathit{final}\ e'$.

Before we discuss the proof we should say a few words about the precondition *wf-J-prog* P that requires P to be a well-formed Jinja program. Its full definition is given in §6.3 below. For ill-formed programs the big and small step semantics of method calls may disagree for the following subtle reason. Big step evaluation needs just the method body and the parameter names *pns* whereas small step reduction also needs the parameter types *Ts* to build the nested blocks because variables in blocks are typed. In an ill-formed program, *Ts* can be shorter than *pns* (well-formedness requires them to be of the same length). Then *blocks* (*this* # *pns*, *Class D* # *Ts*, *Addr a* # *vs*, *body*) loses some parameters because there are no corresponding types. In contrast, the big step semantics will have all parameters in the store l_2(*this* # *pns* $[\mapsto]$ *Addr a* # *vs*).

It is interesting to note that this is the consequence of our choice to separate the parameter names and their types into two lists. In most programming languages they are combined into one list of pairs, thus enforcing automatically that there are as many parameter names as parameter types. We have opted for two lists because it avoids additional operations for splitting a list of pairs.

One half of the only-if-direction of Theorem 5.1 is Lemma 4.1, the other half

Theorem 5.2 If *wf-J-prog P* and $P \vdash \langle e,s\rangle \Rightarrow \langle e',s'\rangle$ then $P \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle$.

is proved by induction on $\Rightarrow$: it is shown that every big step evaluation can be simulated by small step reductions. This requires a number of simple lemmas that lift the subexpression reduction rules from $\rightarrow$ to $\rightarrow^*$. For example, in order to simulate

$$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{null},s_1\rangle \Longrightarrow P \vdash \langle \mathit{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \mathit{null},s_1\rangle$$

we need to prove

$$P \vdash \langle e,s\rangle \rightarrow^* \langle \mathit{null},s'\rangle \Longrightarrow P \vdash \langle \mathit{Cast}\ C\ e,s\rangle \rightarrow^* \langle \mathit{null},s'\rangle$$

which follows from rule $P \vdash \langle \mathit{Cast}\ C\ \mathit{null},s\rangle \rightarrow \langle \mathit{null},s\rangle$ with the help of the lemma

$$P \vdash \langle e,s\rangle \rightarrow^* \langle e',s'\rangle \Longrightarrow P \vdash \langle \mathit{Cast}\ C\ e,s\rangle \rightarrow^* \langle \mathit{Cast}\ C\ e',s'\rangle$$

which is proved from rule

$$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle \mathit{Cast}\ C\ e,s\rangle \rightarrow \langle \mathit{Cast}\ C\ e',s'\rangle$$

by induction on $\rightarrow^*$.

The only non-trivial case is the method call, which is dealt with quite differently in the two semantics. Although we cannot discuss the full proof we want to show the main mediating lemma:

Lemma 5.3 $[\![\mathit{length}\ \mathit{Vs} = \mathit{length}\ \mathit{Ts};\ \mathit{length}\ \mathit{vs} = \mathit{length}\ \mathit{Ts};\ \mathit{distinct}\ \mathit{Vs};$
$P \vdash \langle e,(h,\ l(\mathit{Vs}\ [\mapsto]\ \mathit{vs}))\rangle \rightarrow^* \langle e',(h',\ l')\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{blocks}\ (\mathit{Vs},\ \mathit{Ts},\ \mathit{vs},\ e),(h,\ l)\rangle \rightarrow^*$
$\langle \mathit{blocks}\ (\mathit{Vs},\ \mathit{Ts},\ \mathit{map}\ (\mathit{the} \circ l')\ \mathit{Vs},\ e'),(h',\ l'(l|\mathit{set}\ \mathit{Vs}))\rangle$

It lifts the reduction of the body of a nested block to the whole block. The assumptions about length and distinctness are later discharged with the help of the well-formedness of the program. Distinctness could be dispensed with but then *map* $(\mathit{the} \circ l')$ *Vs* (where *the* is the inverse of *Some*) has to be replaced by a more complex computation.

The other direction of Theorem 5.1

Theorem 5.4 If *wf-J-prog* P and $P \vdash \langle e,s \rangle \rightarrow^* \langle e',s' \rangle$ and *final* e' then $P \vdash \langle e,s \rangle \Rightarrow \langle e',s' \rangle$.

is proved by a trivial induction on (the length of) $\rightarrow^*$. The induction step follows directly from the following key lemma

Theorem 5.5 If *wf-J-prog* P and $P \vdash \langle e,s \rangle \rightarrow \langle e'',s'' \rangle$ and $P \vdash \langle e'',s'' \rangle \Rightarrow \langle e',s' \rangle$ then $P \vdash \langle e,s \rangle \Rightarrow \langle e',s' \rangle$.

which is proved by induction on $\rightarrow$. Surprisingly, we only need two further lemmas in its proof: a big step analogue to the reduction rule for *While*

Lemma 5.6 $P \vdash \langle \mathit{While}\ (b)\ c,s \rangle \Rightarrow \langle e',s' \rangle$ iff $P \vdash \langle \mathit{If}\ (b)\ (c;\ \mathit{While}\ (b)\ c)\ \mathit{Else}\ \mathit{Val}\ \mathit{Unit},s \rangle \Rightarrow \langle e',s' \rangle$

and the converse of Lemma 5.3:

Lemma 5.7 If $\mathit{length}\ ps = \mathit{length}\ ts$ and $\mathit{length}\ ps = \mathit{length}\ vs$ and $P \vdash \langle \mathit{blocks}\ (ps,\ ts,\ vs,\ e),(h,\ l) \rangle \Rightarrow \langle e',(h',\ l') \rangle$ then $\exists\, l''.\ P \vdash \langle e,(h,\ l(ps\ [\mapsto]\ vs)) \rangle \Rightarrow \langle e',(h',\ l'') \rangle \wedge l' = l''(l|\mathit{set}\ ps)$.

It lifts the evaluation of a nested block to the evaluation of its body.

Although the above equivalence proof may appear to be a routine exercise, it could not be lifted directly from the literature: two standard textbooks [17, 5] prove the analogue of Theorem 5.4 for a simple while-language by induction principles that break down for Jinja.

6 Well-Formedness

First we define a type system for expressions, then a check that expression evaluation does not access uninitialized variables, and finally both are used in the definition of well-formed programs.

6.1 Type System

Types are either primitive (*BOOL* and *INTEGER*), class types *Class C*, *NULL* (the type of *Null*), or *VOID* (the type of *Unit*). A **reference type** is either *Class C* or *NULL*; the type *RefT T* subsumes both alternatives.

Function *typeof* :: *heap* ⇒ *val* ⇒ *ty option* computes the type of a value. The heap is necessary because values may contain addresses. The result type is *ty option* rather than *ty* because unallocated addresses do not have a type.

$\mathit{typeof}\ h\ \mathit{Unit} = \mathit{Some}\ \mathit{VOID}$
$\mathit{typeof}\ h\ \mathit{Null} = \mathit{Some}\ \mathit{NULL}$
$\mathit{typeof}\ h\ (\mathit{Bool}\ b) = \mathit{Some}\ \mathit{BOOL}$
$\mathit{typeof}\ h\ (\mathit{Intg}\ i) = \mathit{Some}\ \mathit{INTEGER}$
$\mathit{typeof}\ h\ (\mathit{Addr}\ a) =$
$(\textbf{\textit{case}}\ h\ a\ \textbf{\textit{of}}\ \mathit{None} \Rightarrow \mathit{None}\ |\ \mathit{Some}\ (C,\ \mathit{fs}) \Rightarrow \mathit{Some}\ (\mathit{Class}\ C))$

If we want to rule out addresses in values, thus restricting to "literals", we simply supply an empty heap and define the abbreviation

$$\textit{typeof-lit } v \quad \equiv \quad \textit{typeof empty } v$$

The subclass relationship $P \vdash C \preceq_C C'$ induces a **subtype** relationship $P \vdash T \preceq T'$ in the obvious manner:

$$P \vdash T \preceq T \qquad P \vdash \textit{NULL} \preceq \textit{Class } C$$
$$P \vdash C \preceq_C D \Longrightarrow P \vdash \textit{Class } C \preceq \textit{Class } D$$

The canonical extension of $\preceq$ to lists of types is written $[\preceq]$.

The core of the type system is the judgement $P,E \vdash_a e :: T$, where E is an **environment**, i.e. a map from variables to their types. The complete set of typing rules is shown in Fig. 1. We only discuss the more interesting ones, starting with field access and field assignment. Their typing rules do not just enforce that the types fit together but also that the annotation $\{D\}$ is correct: $\{D\}$ must be the defining class of the field F visible from the static class of the object. (The $_a$ in $\vdash_a$ stands for *a*nnotation.) Alternatively these rules can be viewed as computing $\{D\}$ from P, C and F, namely via *sees-field*. A more sophisticated model (e.g. [2, 3]) of this situation distinguishes annotated from unannotated expressions and has elaboration rules of the form $P,E \vdash e \leadsto e'::T$ where e is the unannotated input expression, e' its annotated variant, and T its type.

Now we examine the remaining rules. We only allow up and down casts: other casts are pointless because they are bound to fail at runtime. Equality comparison ($\ll\!Eq\!\gg$) is allowed between two types if one is a subtype of the other, which of course includes type equality. Loops are of type *VOID* because they evaluate to *Unit*. Exceptions (*Throw*) are of every type, which enables them to occur in any context. The extension of :: to lists is denoted by [::].

Note that for simplicity the rules for $\textit{If } (e)\ e_1 \textit{ Else } e_2$ and $\textit{Try } e_1 \textit{ Catch}(C\ V)\ e_2$ require e_1 and e_2 to have the same type, although it suffices if they have a common supertype. Luckily, equality of types can be arranged by inserting suitable *Cast*s.

6.2 Definite Initialization

One of Java's notable features is the check that all variables must be initialized before use, called "definite assignment". Schirmer [12] has modelled this feature in full detail, with all the special cases that Java prescribes. Jinja's rules for definite initialization are much simpler, thus missing certain cases, but still demonstrating the feature in its full generality. The judgement is $I \triangleright e \triangleright I'$ where I and I' are sets of variables. It captures the following intuition: if all variables in I are initialized before the evaluation of e, the

$\textit{is-class } P\ C \Longrightarrow P,E \vdash_a \textit{New } C :: \textit{Class } C$

$[\![P,E \vdash_a e :: \textit{Class } D;\ \textit{is-class } P\ C;\ P \vdash C \preceq_C D \vee P \vdash D \preceq_C C]\!]$
$\Longrightarrow P,E \vdash_a \textit{Cast } C\ e :: \textit{Class } C$

$\textit{typeof-lit } v = \textit{Some } T \Longrightarrow P,E \vdash_a \textit{Val } v :: T$

$E\ v = \textit{Some } T \Longrightarrow P,E \vdash_a \textit{Var } v :: T$

$[\![P,E \vdash_a e_1 :: T_1;\ P,E \vdash_a e_2 :: T_2;$
$\quad \textbf{\textit{case}}\ \textit{bop}\ \textbf{\textit{of}}\ \textit{Eq} \Rightarrow (P \vdash T_1 \preceq T_2 \vee P \vdash T_2 \preceq T_1) \wedge T = \textit{BOOL}$
$\quad \mid \textit{Add} \Rightarrow T_1 = \textit{INTEGER} \wedge T_2 = \textit{INTEGER} \wedge T = \textit{INTEGER}]\!]$
$\Longrightarrow P,E \vdash_a e_1 \ll\!\textit{bop}\!\gg e_2 :: T$

$[\![P,E \vdash_a \textit{Var } V :: T;\ P,E \vdash_a e :: T';\ P \vdash T' \preceq T;\ V \neq \textit{this}]\!]$
$\Longrightarrow P,E \vdash_a V{:=}e :: T'$

$[\![P,E \vdash_a e :: \textit{Class } C;\ P \vdash C \textit{ sees-field } F{:}T \textit{ in } D]\!] \Longrightarrow P,E \vdash_a \{D\}e{\cdot}F :: T$

$[\![P,E \vdash_a e_1 :: \textit{Class } C;\ P \vdash C \textit{ sees-field } F{:}T \textit{ in } D;\ P,E \vdash_a e_2 :: T';$
$\quad P \vdash T' \preceq T]\!]$
$\Longrightarrow P,E \vdash_a \{D\}e_1{\cdot}F{:=}e_2 :: T'$

$P,E(V \mapsto T) \vdash_a e :: T' \Longrightarrow P,E \vdash_a \{V{:}T;\ e\} :: T'$

$[\![P,E \vdash_a e :: \textit{Class } C;\ P \vdash C \textit{ sees-method } M{:}\ \textit{pTs}' {\rightarrow} \textit{rT} = (\textit{pns},\ \textit{body}) \textit{ in } D;$
$\quad P,E \vdash_a \textit{ps}\ [::]\ \textit{pTs};\ P \vdash \textit{pTs}\ [\preceq]\ \textit{pTs}']\!]$
$\Longrightarrow P,E \vdash_a e{\cdot}M(\textit{ps}) :: \textit{rT}$

$[\![P,E \vdash_a e_1 :: T_1;\ P,E \vdash_a e_2 :: T_2]\!] \Longrightarrow P,E \vdash_a e_1;\ e_2 :: T_2$

$[\![P,E \vdash_a e :: \textit{BOOL};\ P,E \vdash_a e_1 :: T;\ P,E \vdash_a e_2 :: T]\!]$
$\Longrightarrow P,E \vdash_a \textit{If } (e)\ e_1 \textit{ Else } e_2 :: T$

$[\![P,E \vdash_a e :: \textit{BOOL};\ P,E \vdash_a c :: T]\!] \Longrightarrow P,E \vdash_a \textit{While } (e)\ c :: \textit{VOID}$

$P,E \vdash_a e :: \textit{Class } C \Longrightarrow P,E \vdash_a \textit{Throw } e :: T$

$[\![P,E \vdash_a e_1 :: T;\ P,E(V \mapsto \textit{Class } C) \vdash_a e_2 :: T]\!]$
$\Longrightarrow P,E \vdash_a \textit{Try } e_1 \textit{ Catch}(C\ V)\ e_2 :: T$

$P,E \vdash_a []\ [::]\ []$

$[\![P,E \vdash_a e :: T;\ P,E \vdash_a \textit{es}\ [::]\ \textit{Ts}]\!] \Longrightarrow P,E \vdash_a e \# \textit{es}\ [::]\ T \# \textit{Ts}$

Figure 1: Well-typing of input expressions

evaluation will only access initialized variables, and afterwards all variables in I' will be initialized, provided no exception is thrown. Thus it is both a check (no uninitialized variables are accessed) and a computation (of I'). The definition is shown in Fig. 2.

$I_0 \rhd \mathit{New}\ C \rhd I_1 = (I_1 = I_0)$
$I_0 \rhd \mathit{Cast}\ C\ e \rhd I_1 = I_0 \rhd e \rhd I_1$
$I_0 \rhd \mathit{Val}\ v \rhd I_1 = (I_1 = I_0)$
$I_0 \rhd e_1 \ll\!\mathit{bop}\!\gg e_2 \rhd I_2 = (\exists I_1.\ I_0 \rhd e_1 \rhd I_1 \wedge I_1 \rhd e_2 \rhd I_2)$
$I_0 \rhd \mathit{Var}\ V \rhd I_1 = (V \in I_0 \wedge I_1 = I_0)$
$I_0 \rhd V{:=}e \rhd I_2 = (\exists I_1.\ I_0 \rhd e \rhd I_1 \wedge I_2 = I_1 \cup \{V\})$
$I_0 \rhd \{C\}e{\cdot}F \rhd I_1 = I_0 \rhd e \rhd I_1$
$I_0 \rhd \{C\}e_1{\cdot}F{:=}e_2 \rhd I_2 = (\exists I_1.\ I_0 \rhd e_1 \rhd I_1 \wedge I_1 \rhd e_2 \rhd I_2)$
$I_0 \rhd e{\cdot}m(es) \rhd I_2 = (\exists I_1.\ I_0 \rhd e \rhd I_1 \wedge I_1 \rhd[es]\rhd I_2)$
$I_0 \rhd \{V{:}T;\ e\} \rhd I_2 =$
$(\exists I_1.\ I_0 - \{V\} \rhd e \rhd I_1 \wedge$
$\quad I_2 = (\textbf{\textit{if}}\ V \in I_0\ \textbf{\textit{then}}\ I_1 \cup \{V\}\ \textbf{\textit{else}}\ I_1 - \{V\}))$
$I_0 \rhd e_1;\ e_2 \rhd I_2 = (\exists I_1.\ I_0 \rhd e_1 \rhd I_1 \wedge I_1 \rhd e_2 \rhd I_2)$
$I_0 \rhd \mathit{If}\ (e)\ e_1\ \mathit{Else}\ e_2 \rhd I_3 =$
$(\exists I_0'\ I_1\ I_2.$
$\quad I_0 \rhd e \rhd I_0' \wedge I_0' \rhd e_1 \rhd I_1 \wedge I_0' \rhd e_2 \rhd I_2 \wedge I_3 = I_1 \cap I_2)$
$I_0 \rhd \mathit{While}\ (b)\ c \rhd I_1 = (I_0 \rhd b \rhd I_1 \wedge (\exists I_2.\ I_1 \rhd c \rhd I_2))$
$I_0 \rhd \mathit{Throw}\ e \rhd I_2 = (I_2 = \mathit{UNIV} \wedge (\exists I_1.\ I_0 \rhd e \rhd I_1))$
$I_0 \rhd \mathit{Try}\ e_1\ \mathit{Catch}(C\ V)\ e_2 \rhd I_3 =$
$(\exists I_1\ I_2.$
$\quad I_0 \rhd e_1 \rhd I_1 \wedge$
$\quad I_0 \cup \{V\} \rhd e_2 \rhd I_2 \wedge$
$\quad I_3 = I_1 \cap (\textbf{\textit{if}}\ V \in I_0\ \textbf{\textit{then}}\ I_2\ \textbf{\textit{else}}\ I_2 - \{V\}))$
$I_0 \rhd[[]]\rhd I_1 = (I_1 = I_0)$
$I_0 \rhd[e\ \#\ es]\rhd I_2 = (\exists I_1.\ I_0 \rhd e \rhd I_1 \wedge I_1 \rhd[es]\rhd I_2)$

Figure 2: Definit initialization

For a change we have used recursion rather than induction because equalities are simpler to work with than implications. This is possible because the definition is primitive recursive over the syntax, in contrast to the rules for the semantics. We could have defined $\vdash_a$ recursively as well but stuck to the more conventional inductive format. However, behind the scenes we derived the corresponding equations as lemmas to simplify proofs.

The rules for $I \rhd e \rhd I'$ are of a directional nature: one could compute I' from I and e. But it would be a partial function because, for example, there is no I' such that $\{\} \rhd \mathit{Var}\ V \rhd I'$. Thus one would have to define a function of type *vname set* $\Rightarrow$ *expr* $\Rightarrow$ *vname set option*. It is doubtful whether that alternative is any simpler.

Most of the rules are straightforward, but a few deserve some explanations. The rule for $I_0 \rhd \{V{:}T;\ e\} \rhd I_2$ computes an intermediate set I_1 of variables initialized after the evaluation of e starting from $I_0 - \{V\}$;

we must subtract V because it is local and uninitialized. I_1 is almost the correct set of variables initialized after the block. But it may contain V, whereas outside the block, V is only initialized if it was already initialized before entry of the block. Thus we have to remove V from I_1 if it was not in I_0 already.

In a conditional and a loop, the evaluation of the condition has to be taken into account as it may initialize further variables due to side effects. But we are very conservative and do not try to analyze if the condition evaluates to a fixed value. Thus we work with the intersection of the two branches of the conditional and we ignore the effect of the body of the loop (in case it is never entered), although we insist that it passes the definite initialization test (in case it is entered).

The rule for $I_0 \triangleright \mathit{Throw}\ e \triangleright I_2$ may surprise because it equates I_2 with *UNIV*, the set of all variable names.[2] The reason is that definite initialization only guarantees something for normal evaluations. Thus we can be as optimistic as possible (*UNIV*!) for exceptional evaluations (*Throw*). This ensures that in the combination of normal and exceptional behaviour, e.g. in *If* (b) $V{:=}\mathit{null}$ *Else Throw* e, the latter does not interfere with the former.

The rule for *Try-Catch* is a combination of the ones for *If* (because there are two possible execution paths) and blocks (because there is a local variable).

The correctness theorem

Theorem 6.1 If $P \vdash \langle e,(h_0,\ l_0)\rangle \Rightarrow \langle \mathit{Val}\ v,(h_1,\ l_1)\rangle$ and $\mathit{dom}\ l_0 \triangleright e \triangleright I$ then $I \subseteq \mathit{dom}\ l_1$.

is proved by induction on the big step semantics. Note that it only tells us something about evaluations that end in a value, not an exception. The proof relies on the following fairly straightforward monotonicity lemma

If $I_0 \triangleright e \triangleright I_1$ and $I_0 \subseteq I_0'$ then $\exists I_1'.\ I_0' \triangleright e \triangleright I_1' \wedge I_1 \subseteq I_1'$.

which is proved by induction on $I_0 \triangleright e \triangleright I_1$.

The reader should bear in mind that the definite initialization check is only a conservative approximation of the real computation and may reject programs although they are perfectly safe, for example $\{\} \triangleright$ *If* (true) *null Else Var* $V \triangleright \{\}$. But it is good enough for practical purposes, in particular because the programmer can always insert a dummy initialization at the beginning to satisfy the analysis. Thus definite initialization is a mandatory well-formedness property in Jinja and Java.

6.3 Well-Formed Programs

The proposition *wf-J-prog P* formalizes well-formedness of Jinja programs. This requires two global structural properties which we do not formalize

[2] $\mathit{UNIV} \equiv \{x.\ \mathit{True}\}$

here:

- The subclass relationship must be acyclic.
- Method overriding must by contravariant in the arguments (the argument types of the overriding method must be supertypes of the overridden one) and covariant in the result (the result type of the overridden method must be a supertype of the overriding one).

In addition each definition of a method M with list of parameter types pTs, return type rT, list of parameter names *pns* and body *body* in class C in program P is well-formed iff there are as many parameter types as parameter names, the parameter names are distinct, *this* is not among the parameter names, and the method body has a subtype of rT and the definite initialization check succeeds:

$$\begin{array}{l}
\textit{wf-J-mdecl}\ P\ C\ (M,\ pTs,\ rT,\ pns,\ body) = \\
(\textit{length}\ pTs = \textit{length}\ pns\ \wedge \\
\ \textit{distinct}\ pns\ \wedge \\
\ \textit{this} \notin \textit{set}\ pns\ \wedge \\
\ (\exists\, T.\ P,[\textit{this} \mapsto \textit{Class}\ C,\ pns\ [\mapsto]\ pTs] \vdash_a \textit{body} :: T\ \wedge\ P \vdash T \preceq rT)\ \wedge \\
\ (\exists\, I.\ \textit{set}\ pns \cup \{\textit{this}\} \rhd \textit{body} \rhd I))
\end{array}$$

where $[\textit{this} \mapsto \textit{Class}\ C,\ pns\ [\mapsto]\ pTs]$ abbreviates $\textit{empty}(\textit{this}\#pns\ [\mapsto]\ \textit{Class}\ C\#pTs)$, the environment where *this* has type *Class* C and each parameter in *pns* has the corresponding type in pTs.

7 Type Safety

We have proved type safety in the traditional syntactic way [18]: we show **progress** (every well-typed expression that is not *final* can reduce) and **subject reduction** (well typed expressions reduce to well-typed expressions and their type may only become more specific). This requires the following concepts:

$\boldsymbol{P,h \vdash v :: \preceq T}$ (value v **conforms** to type T):

$$\exists\, T'.\ \textit{typeof}\ h\ v = \textit{Some}\ T'\ \wedge\ G \vdash T' \preceq T$$

$\boldsymbol{P \vdash h \surd}$ (**heap conformance**): all objects in h have exactly those fields required by their class (and superclasses) and the value of each field conforms to the declared type. Additionally, all system exceptions (*NullPointer*, *ClassCast* and *OutOfMemory*) are pre-allocated in h.

$\boldsymbol{P,h \vdash l\ (:: \preceq)_w\ E}$: the store l **weakly conforms** to the environment E — weakly because E is allowed to assign a type to a variable that has no value in l, but not the other way around. Formally:

$$\forall\, V\ v.\ l\ V = \textit{Some}\ v \longrightarrow (\exists\, T.\ E\ V = \textit{Some}\ T\ \wedge\ P,h \vdash v :: \preceq T)$$

7.1 Runtime Type System

The proof of subject reduction requires a modified type system. The purpose of $\vdash_a$ is to rule out not just unsafe input expressions but ill-formed ones in general. For example, assignments to *this* are considered bad style and are thus ruled out although such assignments are perfectly safe (and are in fact allowed in the JVM). But now we need a type system that is just strong enough to characterize absence of type safety violations and is invariant under reduction. For a start, during reduction expressions containing addresses may arise. To make them well-typed, the runtime type system takes the heap into account as well (to look up the class of an object) and is written $P,E,h \vdash e :: T$. But there are more subtle changes exemplified by the rules for field access (see §6.1): $P,E \vdash_a \{D\}e{\cdot}F :: T$ requires $P \vdash C$ *sees-field* $F{:}T$ *in* D. However, once e is reduced, its class C may decrease and this condition may no longer be met. Thus we relax it to $P \vdash C$ *has-field* $F{:}T$ *in* D. This is strong enough to guarantee type safety but weak enough to be preserved by reduction. It is interesting to note that this change was missed in [3] (which invalidates their Lemma 6 and thus subject reduction).

The full set of typing rules is given in Fig. 3 (except for the obvious rules for [::]). We discuss only those that differ from their $\vdash_a$-counterpart beyond the addition of h.

A frequent phenomenon is the following. Expression e in $\{D\}e{\cdot}F$, $\{D\}e{\cdot}F{:=}e_2$ and $e{\cdot}M(es)$ is required to be of type *Class* C in input expressions, thus ruling out *null*. However, e may reduce to *null*. Thus we add rules for the case $e :: \mathit{NULL}$. A similar situation arises with *Throw* e and *Cast* C e where we avoid an additional rule by requiring e to be of reference type (which includes *NULL*).

Casts now merely require the expression to be of reference type: during reduction, an initial class type may turn into *NULL*, and an initial down cast may turn into a "sideways" cast — the latter will eventually throw *ClassCast*, which cannot be avoided statically. Equality comparison ($\ll\!\mathit{Eq}\!\gg$) is allowed between arbitrary values because, due to reduction, the type of one side may no longer be a subtype of the other side, and because it does not endanger type safety. In assignments $V{:=}e$ we have dropped the requirement $V \neq \mathit{this}$ just to show that it is irrelevant for type safety. Typing of field assignment has changed in analogy with field access. Typing *Try* e_1 *Catch*$(C\ V)$ e_2 no longer requires e_1 and e_2 to be of the same type: during reduction of e_1 its type may become a subtype of that of e_2.

As a sanity check we can prove that the runtime type system is no more restrictive than the one for input expressions:

Lemma 7.1 If $P,E \vdash_a e :: T$ then $P,E,h \vdash e :: T$.

The proof is by induction on $\vdash_a$.

$\textit{is-class } P\ C \Longrightarrow P{,}E{,}h \vdash \textit{New } C :: \textit{Class } C$

$[\![P{,}E{,}h \vdash e :: \textit{RefT } T;\ \textit{is-class } P\ C]\!] \Longrightarrow P{,}E{,}h \vdash \textit{Cast } C\ e :: \textit{Class } C$

$\textit{typeof } h\ v = \textit{Some } T \Longrightarrow P{,}E{,}h \vdash \textit{Val } v :: T$

$E\ v = \textit{Some } T \Longrightarrow P{,}E{,}h \vdash \textit{Var } v :: T$

$[\![P{,}E{,}h \vdash e_1 :: T_1;\ P{,}E{,}h \vdash e_2 :: T_2;$
$\quad \textbf{\textit{case}}\ \textit{bop}\ \textbf{\textit{of}}\ \textit{Eq} \Rightarrow T' = \textit{BOOL}$
$\quad \mid \textit{Add} \Rightarrow T_1 = \textit{INTEGER} \wedge T_2 = \textit{INTEGER} \wedge T' = \textit{INTEGER}]\!]$
$\Longrightarrow P{,}E{,}h \vdash e_1 \ll\!\textit{bop}\!\gg e_2 :: T'$

$[\![P{,}E{,}h \vdash \textit{Var } V :: T;\ P{,}E{,}h \vdash e :: T';\ P \vdash T' \preceq T]\!] \Longrightarrow P{,}E{,}h \vdash V{:=}e :: T'$

$[\![P{,}E{,}h \vdash e :: \textit{Class } C;\ P \vdash C \textit{ has-field } F{:}T \textit{ in } D]\!] \Longrightarrow P{,}E{,}h \vdash \{D\}e{\cdot}F :: T$

$P{,}E{,}h \vdash e :: \textit{NULL} \Longrightarrow P{,}E{,}h \vdash \{D\}e{\cdot}F :: T$

$[\![P{,}E{,}h \vdash e_1 :: \textit{Class } C;\ P \vdash C \textit{ has-field } F{:}T \textit{ in } D;\ P{,}E{,}h \vdash e_2 :: T_2;$
$\quad P \vdash T_2 \preceq T]\!]$
$\Longrightarrow P{,}E{,}h \vdash \{D\}e_1{\cdot}F{:=}e_2 :: T_2$

$[\![P{,}E{,}h \vdash e_1 :: \textit{NULL};\ P{,}E{,}h \vdash e_2 :: T_2]\!] \Longrightarrow P{,}E{,}h \vdash \{D\}e_1{\cdot}F{:=}e_2 :: T$

$P{,}E(V \mapsto T){,}h \vdash e :: T' \Longrightarrow P{,}E{,}h \vdash \{V{:}T;\ e\} :: T'$

$[\![P{,}E{,}h \vdash e :: \textit{Class } C;\ P \vdash C \textit{ sees-method } M{:}\ \textit{Ts}{\rightarrow}T = (\textit{pns},\ \textit{body}) \textit{ in } D;$
$\quad P{,}E{,}h \vdash \textit{es}\ [::]\ \textit{Ts}';\ P \vdash \textit{Ts}'\ [\preceq]\ \textit{Ts}]\!]$
$\Longrightarrow P{,}E{,}h \vdash e{\cdot}M(\textit{es}) :: T$

$[\![P{,}E{,}h \vdash e :: \textit{NULL};\ P{,}E{,}h \vdash \textit{es}\ [::]\ \textit{Ts}]\!] \Longrightarrow P{,}E{,}h \vdash e{\cdot}M(\textit{es}) :: T$

$[\![P{,}E{,}h \vdash e_1 :: T_1;\ P{,}E{,}h \vdash e_2 :: T_2]\!] \Longrightarrow P{,}E{,}h \vdash e_1;\ e_2 :: T_2$

$[\![P{,}E{,}h \vdash e :: \textit{BOOL};\ P{,}E{,}h \vdash e_1 :: T;\ P{,}E{,}h \vdash e_2 :: T]\!]$
$\Longrightarrow P{,}E{,}h \vdash \textit{If } (e)\ e_1 \textit{ Else } e_2 :: T$

$[\![P{,}E{,}h \vdash e :: \textit{BOOL};\ P{,}E{,}h \vdash c :: T]\!] \Longrightarrow P{,}E{,}h \vdash \textit{While } (e)\ c :: \textit{VOID}$

$P{,}E{,}h \vdash e :: \textit{RefT } R \Longrightarrow P{,}E{,}h \vdash \textit{Throw } e :: T$

$[\![P{,}E{,}h \vdash e_1 :: T_1;\ P{,}E(V \mapsto \textit{Class } C){,}h \vdash e_2 :: T_2;\ P \vdash T_1 \preceq T_2]\!]$
$\Longrightarrow P{,}E{,}h \vdash \textit{Try } e_1\ \textit{Catch}(C\ V)\ e_2 :: T_2$

Figure 3: Runtime type system

7.2 Type Safety Proof

7.2.1 Progress

Under suitable conditions we can now show progress:

Lemma 7.2 If *wf-J-prog* P and $P,E,h \vdash e :: T$ and $P \vdash h \surd$ and *dom* $l \triangleright e \triangleright I$ and $\neg$ *final* e then $\exists\, e'\ h'\ l'.\ P \vdash \langle e,(h,\ l)\rangle \rightarrow \langle e',(h',\ l')\rangle$.

The proof is by induction on $P,E,h \vdash e :: T$.

Let us examine the necessity for the individual premises. Well-formedness of P is necessary for the following subtle reason: even if P defines a class C, relations $P \vdash C$ *has-fields* ... (needed for the reduction of *New*) and $P \vdash C$ *sees-method* ... (needed for reduction of method calls) are only defined if P is well-formed because acyclicity is needed in the traversal of the class hierarchy. Well-typedness of e is needed, for example, to ensure that in every method call the number of formal and actual parameters agrees. Heap conformance ($P \vdash h \surd$) is needed because otherwise an object may not have all the fields of its class and field access may get stuck. Definite initialization is required to ensure that variable access does not get stuck.

The proof of Lemma 7.2 is in fact again a simultaneous inductive proof of the above statement and the corresponding one for expressions lists, where $[::]$, $[\rightarrow]$ and *finals* replace $::$, $\rightarrow$ and *final*.

7.2.2 Preservation Theorems

Eventually we show that a sequence of reductions preserves well-typedness by showing that each reduction step preserves well-typedness. However, well-typedness is not preserved on its own but requires additional assumptions, e.g. conformance of the initial heap. Thus we need to show conformance of all intermediate heaps, i.e. preservation of heap conformance with each step. In total we need three auxiliary preservation theorems which are all proved by induction on $P \vdash \langle e,(h,\ l)\rangle \rightarrow \langle e',(h',\ l')\rangle$:

Theorem 7.3 If $P \vdash \langle e,(h,\ l)\rangle \rightarrow \langle e',(h',\ l')\rangle$ and $P,E,h \vdash e :: T$ and $P \vdash h \surd$ then $P \vdash h' \surd$.

Theorem 7.4 If $P \vdash \langle e,(h,\ l)\rangle \rightarrow \langle e',(h',\ l')\rangle$ and $P,E,h \vdash e :: T$ and $P,h \vdash l\ (::\preceq)_w\ E$ then $P,h' \vdash l'\ (::\preceq)_w\ E$.

Theorem 7.5 If *wf-J-prog* P and $P \vdash \langle e,(h,\ l)\rangle \rightarrow \langle e',(h',\ l')\rangle$ and *dom* $l \triangleright e \triangleright I$ then $\exists I'.$ *dom* $l' \triangleright e' \triangleright I' \wedge I \subseteq I'$.

The last of these is somewhat more complex to prove than the others.

7.2.3 Subject Reduction

The core of the proof is the single step subject reduction theorem:

Theorem 7.6 If *wf-J-prog* P and $P \vdash \langle e,(h,\ l)\rangle \rightarrow \langle e',(h',\ l')\rangle$ and $P \vdash h\ \surd$ and $P,h \vdash l\ (::\preceq)_w\ E$ and $P,E,h \vdash e :: T$ then $\exists\, T'.\ P,E,h' \vdash e' :: T' \wedge P \vdash T' \preceq T$.

The proof is again by induction on $P \vdash \langle e,(h,\ l)\rangle \rightarrow \langle e',(h',\ l')\rangle$.

Now we extend progress and subject reduction to $\rightarrow^*$. To ease notation we introduce the following definition

$P,E,s \vdash_C e :: T \equiv$
let $(h,\ l) = s$
in $P \vdash h\ \surd \wedge P,h \vdash l\ (::\preceq)_w\ E \wedge (\exists\, I.\ \mathit{dom}\ l \rhd e \rhd I) \wedge P,E,h \vdash e :: T$

where $_C$ stands for "configuration". Now we can rephrase progress more succinctly

$[\![\mathit{wf\text{-}J\text{-}prog}\ P;\ P,E,s \vdash_C e :: T;\ \neg\ \mathit{final}\ e]\!] \Longrightarrow \exists\, e'\ s'.\ P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle$

and we can combine the auxiliary preservation theorems and subject reduction:

$[\![\mathit{wf\text{-}J\text{-}prog}\ P;\ P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle;\ P,E,s \vdash_C e :: T]\!]$
$\Longrightarrow \exists\, T'.\ P,E,s' \vdash_C e' :: T' \wedge P \vdash T' \preceq T$

From these two corollaries an easy induction on $\rightarrow^*$ yields the final form of subject reduction:

Theorem 7.7 If *wf-J-prog* P and $P \vdash \langle e_0,s_0\rangle \rightarrow^* \langle e_1,s_1\rangle$ and $\neg\ (\exists\, e_2\ s_2.\ P \vdash \langle e_1,s_1\rangle \rightarrow \langle e_2,s_2\rangle)$ and $P,E,s_0 \vdash_C e_0 :: T_0$ then *final* $e_1 \wedge (\exists\, T_1.\ P,E,s_1 \vdash_C e_1 :: T_1 \wedge P \vdash T_1 \preceq T_0)$.

In words: if we reduce an expression to a normal form and the initial expression has type T_0, the normal form will be a final expression whose type is a subtype of T_0.

The only "flaw" in the statement of this theorem is that it refers to the runtime type system and not to the one for input expression. Luckily we have Lemma 4.1 and can thus conclude

Corollary 7.8 If *wf-J-prog* P and $P \vdash h_0\ \surd$ and $P,h_0 \vdash l_0\ (::\preceq)_w\ E$ and $\exists\, I.\ \mathit{dom}\ l_0 \rhd e_0 \rhd I$ and $P,E \vdash_a e_0 :: T$ and $P \vdash \langle e_0,(h_0,\ l_0)\rangle \rightarrow^* \langle e_1,(h_1,\ l_1)\rangle$ and $\neg\ (\exists\, e_2\ s_2.\ P \vdash \langle e_1,(h_1,\ l_1)\rangle \rightarrow \langle e_2,s_2\rangle)$ then $(\exists\, v.\ e_1 = \mathit{Val}\ v \wedge P,h_1 \vdash v ::\preceq T) \vee (\exists\, a.\ e_1 = \mathit{THROW}\ a \wedge a \in \mathit{dom}\ h_1)$.

In words: if the initial state is OK and the expression passes the definite initialization check and is well-typed according to the input type system $\vdash_a$, then reduction to normal form yields either a value of a subtype of the initial expression or throws an existing object.

8 Conclusion

This ends the presentation of the source language with its two semantics, its type system, and the proof of type safety. A virtual machine and bytecode verifier have already been formalized [6, 4] and a verified compiler (similar to [14]) is almost completed.

Comparing our language to related work we find that it is closest to [2] but formalizes additional aspects like definite initialization. In particular giving both a small and a big step semantics and relating them appears new. The advantage of having both is that they are suitable for different purposes: the small step semantics for the type safety proof and the big step semantics for a compiler correctness proof [14]. With respect to other machine checked formalizations of Java-like languages we find that our semantics improves on [8] with its awkward treatment of exceptions and on [15] with its explicit frame stack.

Acknowledgements I would like to thank Norbert Schirmer and Daniel Wasserrab for carrying out the proof of Theorem 5.4, Gerwin Klein and Norbert Schirmer for many discussions, and Sophia Drossopoulou for critcal reading and comments.

Appendix

A Evaluation Rules

New:

$[\![\textit{new-Addr}\ h = \textit{Some}\ a;\ P \vdash C\ \textit{has-fields}\ \textit{FDTs};$
$\quad h' = h(a \mapsto (C,\ \textit{init-vars}\ \textit{FDTs}))]\!]$
$\Longrightarrow P \vdash \langle \textit{New}\ C,(h,\ l)\rangle \Rightarrow \langle \textit{addr}\ a,(h',\ l)\rangle$

$\textit{new-Addr}\ h = \textit{None} \Longrightarrow P \vdash \langle \textit{New}\ C,(h,\ l)\rangle \Rightarrow \langle \textit{throw}\ \textit{OutOfMemory},(h,\ l)\rangle$

Cast:

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{addr}\ a,(h,\ l)\rangle;\ h\ a = \textit{Some}\ (D,\ \textit{fs});\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle \textit{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{addr}\ a,(h,\ l)\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{null},s_1\rangle \Longrightarrow P \vdash \langle \textit{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{null},s_1\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{addr}\ a,(h,\ l)\rangle;\ h\ a = \textit{Some}\ (D,\ \textit{fs});\ \neg\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle \textit{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{throw}\ \textit{ClassCast},(h,\ l)\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \textit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle \textit{Cast}\ C\ e,s_0\rangle \Rightarrow \langle \textit{Throw}\ e',s_1\rangle$

Value:

$P \vdash \langle \mathit{Val}\ v,s\rangle \Rightarrow \langle \mathit{Val}\ v,s\rangle$

Binary operation:

$\llbracket P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Val}\ v_1,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Val}\ v_2,s_2\rangle;$
$\quad v = \mathit{binop}\ \mathit{bop}\ v_1\ v_2\rrbracket$
$\Longrightarrow P \vdash \langle e_1 \ll\!\mathit{bop}\!\gg e_2,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_2\rangle$

$P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_1\rangle \Longrightarrow P \vdash \langle e_1 \ll\!\mathit{bop}\!\gg e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_1\rangle$

$\llbracket P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Val}\ v_1,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Throw}\ e,s_2\rangle\rrbracket$
$\Longrightarrow P \vdash \langle e_1 \ll\!\mathit{bop}\!\gg e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_2\rangle$

Variable access:

$l\ V = \mathit{Some}\ v \Longrightarrow P \vdash \langle \mathit{Var}\ V,(h,\ l)\rangle \Rightarrow \langle \mathit{Val}\ v,(h,\ l)\rangle$

Variable assignment:

$\llbracket P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,(h,\ l)\rangle;\ l' = l(V \mapsto v)\rrbracket$
$\Longrightarrow P \vdash \langle V{:=}e,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,(h,\ l')\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle V{:=}e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

Field access:

$\llbracket P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{addr}\ a,(h,\ l)\rangle;\ h\ a = \mathit{Some}\ (C,\ \mathit{fs});\ \mathit{fs}\ (F,\ D) = \mathit{Some}\ v\rrbracket$
$\Longrightarrow P \vdash \langle \{D\}e{\cdot}F,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,(h,\ l)\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{null},s_1\rangle \Longrightarrow P \vdash \langle \{D\}e{\cdot}F,s_0\rangle \Rightarrow \langle \mathit{throw}\ \mathit{NullPointer},s_1\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle \{D\}e{\cdot}F,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

Field assignment:

$\llbracket P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{addr}\ a,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Val}\ v,(h_2,\ l_2)\rangle;$
$\quad h_2\ a = \mathit{Some}\ (C,\ \mathit{fs});\ \mathit{fs}' = \mathit{fs}((F,\ D) \mapsto v);\ h_2' = h_2(a \mapsto (C,\ \mathit{fs}'))\rrbracket$
$\Longrightarrow P \vdash \langle \{D\}e_1{\cdot}F{:=}e_2,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,(h_2',\ l_2)\rangle$

$\llbracket P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{null},s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Val}\ v,s_2\rangle\rrbracket$
$\Longrightarrow P \vdash \langle \{D\}e_1{\cdot}F{:=}e_2,s_0\rangle \Rightarrow \langle \mathit{throw}\ \mathit{NullPointer},s_2\rangle$

$P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle \{D\}e_1{\cdot}F{:=}e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$\llbracket P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle \mathit{Throw}\ e',s_2\rangle\rrbracket$
$\Longrightarrow P \vdash \langle \{D\}e_1{\cdot}F{:=}e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_2\rangle$

Method call:

$\llbracket P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{addr}\ a,s_1\rangle;\ P \vdash \langle \mathit{ps},s_1\rangle\ [\Rightarrow]\ \langle \mathit{map}\ \mathit{Val}\ \mathit{vs},(h_2,\ l_2)\rangle;$
$\quad h_2\ a = \mathit{Some}\ (C,\ \mathit{fs});\ P \vdash C\ \mathit{sees\text{-}method}\ M\colon \mathit{Ts}{\rightarrow}T = (\mathit{pns},\ \mathit{body})\ \mathit{in}\ D;$
$\quad \mathit{length}\ \mathit{vs} = \mathit{length}\ \mathit{Ts};\ l_2' = l_2(\mathit{this} \mapsto \mathit{Addr}\ a,\ \mathit{pns}\ [\mapsto]\ \mathit{vs});$
$\quad P \vdash \langle \mathit{body},(h_2,\ l_2')\rangle \Rightarrow \langle e',(h_3,\ l_3)\rangle;$
$\quad l_3' = l_3(l_2|\{\mathit{this}\} \cup \mathit{set}\ \mathit{pns})\rrbracket$
$\Longrightarrow P \vdash \langle e{\cdot}M(\mathit{ps}),s_0\rangle \Rightarrow \langle e',(h_3,\ l_3')\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{null},s_1\rangle;\ P \vdash \langle ps,s_1\rangle\ [\Rightarrow]\ \langle \mathit{map\ Val\ vs},s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle \mathit{throw\ NullPointer},s_2\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle e{\cdot}M(ps),s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle;$
$\quad es' = \mathit{map\ Val\ vs}\ @\ \mathit{Throw\ ex}\ \#\ es_2]\!]$
$\Longrightarrow P \vdash \langle e{\cdot}M(es),s_0\rangle \Rightarrow \langle \mathit{Throw\ ex},s_2\rangle$

Block:

$P \vdash \langle e_0,(h_0,\ l_0(V := \mathit{None}))\rangle \Rightarrow \langle e_1,(h_1,\ l_1)\rangle \Longrightarrow$
$P \vdash \langle \{V{:}T;\ e_0\},(h_0,\ l_0)\rangle \Rightarrow \langle e_1,(h_1,\ l_1(V := l_0\ V))\rangle$

Sequential composition:

$[\![P \vdash \langle e_0,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash \langle e_1,s_1\rangle \Rightarrow \langle e_2,s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e_0;\ e_1,s_0\rangle \Rightarrow \langle e_2,s_2\rangle$

$P \vdash \langle e_0,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_1\rangle \Longrightarrow P \vdash \langle e_0;\ e_1,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e,s_1\rangle$

Conditional:

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{true},s_1\rangle;\ P \vdash \langle e_1,s_1\rangle \Rightarrow \langle e',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{If}\ (e)\ e_1\ \mathit{Else}\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{false},s_1\rangle;\ P \vdash \langle e_2,s_1\rangle \Rightarrow \langle e',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{If}\ (e)\ e_1\ \mathit{Else}\ e_2,s_0\rangle \Rightarrow \langle e',s_2\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow$
$P \vdash \langle \mathit{If}\ (e)\ e_1\ \mathit{Else}\ e_2,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

While loop:

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{false},s_1\rangle \Longrightarrow P \vdash \langle \mathit{While}\ (e)\ c,s_0\rangle \Rightarrow \langle \mathit{Val\ Unit},s_1\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{true},s_1\rangle;\ P \vdash \langle c,s_1\rangle \Rightarrow \langle \mathit{Val}\ v_1,s_2\rangle;$
$\quad P \vdash \langle \mathit{While}\ (e)\ c,s_2\rangle \Rightarrow \langle e_3,s_3\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{While}\ (e)\ c,s_0\rangle \Rightarrow \langle e_3,s_3\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle \mathit{While}\ (e)\ c,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{true},s_1\rangle;\ P \vdash \langle c,s_1\rangle \Rightarrow \langle \mathit{Throw}\ e',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{While}\ (e)\ c,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_2\rangle$

Throw:

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{addr}\ a,s_1\rangle \Longrightarrow P \vdash \langle \mathit{Throw}\ e,s_0\rangle \Rightarrow \langle \mathit{THROW}\ a,s_1\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{null},s_1\rangle \Longrightarrow P \vdash \langle \mathit{Throw}\ e,s_0\rangle \Rightarrow \langle \mathit{throw\ NullPointer},s_1\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle \mathit{Throw}\ e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle$

Try-Catch:

$P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{Val}\ v_1,s_1\rangle \Longrightarrow$
$P \vdash \langle \mathit{Try}\ e_1\ \mathit{Catch}(C\ V)\ e_2,s_0\rangle \Rightarrow \langle \mathit{Val}\ v_1,s_1\rangle$

$[\![P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{THROW}\ a,(h_1,\ l_1)\rangle;\ h_1\ a = \mathit{Some}\ (D,\ \mathit{fs});\ P \vdash D \preceq_C C;$
$\quad P \vdash \langle e_2,(h_1,\ l_1(V \mapsto \mathit{Addr}\ a))\rangle \Rightarrow \langle e_2{}',(h_2,\ l_2)\rangle]\!]$
$\Longrightarrow P \vdash \langle \mathit{Try}\ e_1\ \mathit{Catch}(C\ V)\ e_2,s_0\rangle \Rightarrow \langle e_2{}',(h_2,\ l_2(V := l_1\ V))\rangle$

$[\![P \vdash \langle e_1,s_0\rangle \Rightarrow \langle \mathit{THROW}\ a,(h_1,\ l_1)\rangle;\ h_1\ a = \mathit{Some}\ (D,\ \mathit{fs});\ \neg\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle \mathit{Try}\ e_1\ \mathit{Catch}(C\ V)\ e_2,s_0\rangle \Rightarrow \langle \mathit{THROW}\ a,(h_1,\ l_1)\rangle$

Expression lists:

$P \vdash \langle [],s\rangle\ [\Rightarrow]\ \langle [],s\rangle$

$[\![P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Val}\ v,s_1\rangle;\ P \vdash \langle es,s_1\rangle\ [\Rightarrow]\ \langle es',s_2\rangle]\!]$
$\Longrightarrow P \vdash \langle e\ \#\ es,s_0\rangle\ [\Rightarrow]\ \langle \mathit{Val}\ v\ \#\ es',s_2\rangle$

$P \vdash \langle e,s_0\rangle \Rightarrow \langle \mathit{Throw}\ e',s_1\rangle \Longrightarrow P \vdash \langle e\ \#\ es,s_0\rangle\ [\Rightarrow]\ \langle \mathit{Throw}\ e'\ \#\ es,s_1\rangle$

B Reduction Rules

New:

$[\![\mathit{new\text{-}Addr}\ h = \mathit{Some}\ a;\ P \vdash C\ \mathit{has\text{-}fields}\ \mathit{FDTs};$
$\quad h' = h(a \mapsto (C,\ \mathit{init\text{-}vars}\ \mathit{FDTs}))]\!]$
$\Longrightarrow P \vdash \langle \mathit{New}\ C,(h,\ l)\rangle \rightarrow \langle \mathit{addr}\ a,(h',\ l)\rangle$

$\mathit{new\text{-}Addr}\ h = \mathit{None} \Longrightarrow P \vdash \langle \mathit{New}\ C,(h,\ l)\rangle \rightarrow \langle \mathit{throw}\ \mathit{OutOfMemory},(h,\ l)\rangle$

Cast:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle \mathit{Cast}\ C\ e,s\rangle \rightarrow \langle \mathit{Cast}\ C\ e',s'\rangle$

$P \vdash \langle \mathit{Cast}\ C\ \mathit{null},s\rangle \rightarrow \langle \mathit{null},s\rangle$

$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (D,\ \mathit{fs});\ P \vdash D \preceq_C C]\!] \Longrightarrow P \vdash \langle \mathit{Cast}\ C\ (\mathit{addr}\ a),s\rangle \rightarrow \langle \mathit{addr}\ a,s\rangle$

$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (D,\ \mathit{fs});\ \neg\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle \mathit{Cast}\ C\ (\mathit{addr}\ a),s\rangle \rightarrow \langle \mathit{throw}\ \mathit{ClassCast},s\rangle$

$P \vdash \langle \mathit{Cast}\ C\ (\mathit{Throw}\ e),s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

Binary operation:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e \ll \mathit{bop} \gg e_2,s\rangle \rightarrow \langle e' \ll \mathit{bop} \gg e_2,s'\rangle$

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle \mathit{Val}\ v_1 \ll \mathit{bop} \gg e,s\rangle \rightarrow \langle \mathit{Val}\ v_1 \ll \mathit{bop} \gg e',s'\rangle$

$v = \mathit{binop}\ \mathit{bop}\ v_1\ v_2 \Longrightarrow P \vdash \langle \mathit{Val}\ v_1 \ll \mathit{bop} \gg \mathit{Val}\ v_2,s\rangle \rightarrow \langle \mathit{Val}\ v,s\rangle$

$P \vdash \langle \mathit{Throw}\ e \ll\!\mathit{bop}\!\gg e_2,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \mathit{Val}\ v_1 \ll\!\mathit{bop}\!\gg \mathit{Throw}\ e,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

Variable access:

$\mathit{lcl}\ s\ V = \mathit{Some}\ v \Longrightarrow P \vdash \langle \mathit{Var}\ V,s\rangle \rightarrow \langle \mathit{Val}\ v,s\rangle$

Variable assignment:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle V{:=}e,s\rangle \rightarrow \langle V{:=}e',s'\rangle$

$P \vdash \langle V{:=}\mathit{Val}\ v,(h,\ l)\rangle \rightarrow \langle \mathit{Val}\ v,(h,\ l(V \mapsto v))\rangle$

$P \vdash \langle V{:=}\mathit{Throw}\ e,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

Field access:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle \{D\}e{\cdot}F,s\rangle \rightarrow \langle \{D\}e'{\cdot}F,s'\rangle$

$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (C,\ \mathit{fs});\ \mathit{fs}\ (F,\ D) = \mathit{Some}\ v]\!]$
$\Longrightarrow P \vdash \langle \{D\}\mathit{addr}\ a{\cdot}F,s\rangle \rightarrow \langle \mathit{Val}\ v,s\rangle$

$P \vdash \langle \{T\}\mathit{null}{\cdot}F,s\rangle \rightarrow \langle \mathit{throw}\ \mathit{NullPointer},s\rangle$

$P \vdash \langle \{T\}\mathit{Throw}\ e{\cdot}F,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

Field assignment:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle \{D\}e{\cdot}F{:=}e_2,s\rangle \rightarrow \langle \{D\}e'{\cdot}F{:=}e_2,s'\rangle$

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle \{D\}\mathit{Val}\ v{\cdot}F{:=}e,s\rangle \rightarrow \langle \{D\}\mathit{Val}\ v{\cdot}F{:=}e',s'\rangle$

$h\ a = \mathit{Some}\ (C,\ \mathit{fs}) \Longrightarrow$
$P \vdash \langle \{D\}\mathit{addr}\ a{\cdot}F{:=}\mathit{Val}\ v,(h,\ l)\rangle \rightarrow \langle \mathit{Val}\ v,(h(a \mapsto (C,\ \mathit{fs}((F,\ D) \mapsto v))),\ l)\rangle$

$P \vdash \langle \{D\}\mathit{null}{\cdot}F{:=}\mathit{Val}\ v,s\rangle \rightarrow \langle \mathit{throw}\ \mathit{NullPointer},s\rangle$

$P \vdash \langle \{D\}\mathit{Throw}\ e{\cdot}F{:=}e_2,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

$P \vdash \langle \{D\}\mathit{Val}\ v{\cdot}F{:=}\mathit{Throw}\ e,s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

Method call:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e{\cdot}M(\mathit{es}),s\rangle \rightarrow \langle e'{\cdot}M(\mathit{es}),s'\rangle$

$P \vdash \langle \mathit{es},s\rangle\ [\rightarrow]\ \langle \mathit{es}',s'\rangle \Longrightarrow P \vdash \langle \mathit{Val}\ v{\cdot}M(\mathit{es}),s\rangle \rightarrow \langle \mathit{Val}\ v{\cdot}M(\mathit{es}'),s'\rangle$

$[\![\mathit{hp}\ s\ a = \mathit{Some}\ (C,\ \mathit{fs});\ P \vdash C\ \mathit{sees\text{-}method}\ M\colon \mathit{Ts} \rightarrow T = (\mathit{pns},\ \mathit{body})\ \mathit{in}\ D;$
$\quad \mathit{length}\ \mathit{vs} = \mathit{length}\ \mathit{Ts}]\!]$
$\Longrightarrow P \vdash \langle \mathit{addr}\ a{\cdot}M(\mathit{map}\ \mathit{Val}\ \mathit{vs}),s\rangle \rightarrow$
$\qquad \langle \mathit{blocks}\ (\mathit{this}\ \#\ \mathit{pns},\ \mathit{Class}\ D\ \#\ \mathit{Ts},\ \mathit{Addr}\ a\ \#\ \mathit{vs},\ \mathit{body}),s\rangle$

$P \vdash \langle null{\cdot}M(map\ Val\ vs),s\rangle \to \langle throw\ NullPointer,s\rangle$

$P \vdash \langle Throw\ e{\cdot}M(es),s\rangle \to \langle Throw\ e,s\rangle$

$es = map\ Val\ vs\ @\ Throw\ e\ \#\ es' \Longrightarrow P \vdash \langle Val\ v{\cdot}M(es),s\rangle \to \langle Throw\ e,s\rangle$

Block:

$[\![P \vdash \langle e,(h,\ l(V := None))\rangle \to \langle e',(h',\ l')\rangle;\ l'\ V = None;\ \neg\ assigned\ V\ e]\!]$
$\Longrightarrow P \vdash \langle \{V{:}T;\ e\},(h,\ l)\rangle \to \langle \{V{:}T;\ e'\},(h',\ l'(V := l\ V))\rangle$

$[\![P \vdash \langle e,(h,\ l(V := None))\rangle \to \langle e',(h',\ l')\rangle;\ l'\ V = Some\ v;\ \neg\ assigned\ V\ e]\!]$
$\Longrightarrow P \vdash \langle \{V{:}T;\ e\},(h,\ l)\rangle \to \langle \{V{:}T;\ V{:=}Val\ v;\ e'\},(h',\ l'(V := l\ V))\rangle$

$[\![P \vdash \langle e,(h,\ l(V \mapsto v))\rangle \to \langle e',(h',\ l')\rangle;\ l'\ V = Some\ v']\!]$
$\Longrightarrow P \vdash \langle \{V{:}T;\ V{:=}Val\ v;\ e\},(h,\ l)\rangle \to$
$\langle \{V{:}T;\ V{:=}Val\ v';\ e'\},(h',\ l'(V := l\ V))\rangle$

$P \vdash \langle \{V{:}T;\ Val\ u\},s\rangle \to \langle Val\ u,s\rangle$

$P \vdash \langle \{V{:}T;\ V{:=}Val\ v;\ Val\ u\},s\rangle \to \langle Val\ u,s\rangle$

$P \vdash \langle \{V{:}T;\ THROW\ a\},s\rangle \to \langle THROW\ a,s\rangle$

$P \vdash \langle \{V{:}T;\ V{:=}Val\ v;\ THROW\ a\},s\rangle \to \langle THROW\ a,s\rangle$

Sequential composition:

$P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow P \vdash \langle e;\ e_2,s\rangle \to \langle e';\ e_2,s'\rangle$

$P \vdash \langle Val\ v;\ e_2,s\rangle \to \langle e_2,s\rangle$

$P \vdash \langle Throw\ e;\ e_2,s\rangle \to \langle Throw\ e,s\rangle$

Conditional:

$P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow$
$P \vdash \langle If\ (e)\ e_1\ Else\ e_2,s\rangle \to \langle If\ (e')\ e_1\ Else\ e_2,s'\rangle$

$P \vdash \langle If\ (true)\ e_1\ Else\ e_2,s\rangle \to \langle e_1,s\rangle$

$P \vdash \langle If\ (false)\ e_1\ Else\ e_2,s\rangle \to \langle e_2,s\rangle$

$P \vdash \langle If\ (Throw\ e)\ e_1\ Else\ e_2,s\rangle \to \langle Throw\ e,s\rangle$

While loop:

$P \vdash \langle While\ (b)\ c,s\rangle \to \langle If\ (b)\ (c;\ While\ (b)\ c)\ Else\ Val\ Unit,s\rangle$

Throw:

$P \vdash \langle e,s\rangle \to \langle e',s'\rangle \Longrightarrow P \vdash \langle Throw\ e,s\rangle \to \langle Throw\ e',s'\rangle$

$P \vdash \langle \mathit{Throw}\ \mathit{null},s\rangle \rightarrow \langle \mathit{throw}\ \mathit{NullPointer},s\rangle$

$P \vdash \langle \mathit{Throw}\ (\mathit{Throw}\ e),s\rangle \rightarrow \langle \mathit{Throw}\ e,s\rangle$

Try-Catch:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow$
$P \vdash \langle \mathit{Try}\ e\ \mathit{Catch}(C\ V)\ e_2,s\rangle \rightarrow \langle \mathit{Try}\ e'\ \mathit{Catch}(C\ V)\ e_2,s'\rangle$

$P \vdash \langle \mathit{Try}\ \mathit{Val}\ v\ \mathit{Catch}(C\ V)\ e_2,s\rangle \rightarrow \langle \mathit{Val}\ v,s\rangle$

$[\![hp\ s\ a = \mathit{Some}\ (D,\ fs);\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle \mathit{Try}\ \mathit{THROW}\ a\ \mathit{Catch}(C\ V)\ e_2,s\rangle \rightarrow \langle\{V{:}\mathit{Class}\ C;\ V{:=}\mathit{addr}\ a;\ e_2\},s\rangle$

$[\![hp\ s\ a = \mathit{Some}\ (D,\ fs);\ \neg\ P \vdash D \preceq_C C]\!]$
$\Longrightarrow P \vdash \langle \mathit{Try}\ \mathit{THROW}\ a\ \mathit{Catch}(C\ V)\ e_2,s\rangle \rightarrow \langle \mathit{THROW}\ a,s\rangle$

Expression list:

$P \vdash \langle e,s\rangle \rightarrow \langle e',s'\rangle \Longrightarrow P \vdash \langle e\ \#\ es,s\rangle\ [\rightarrow]\ \langle e'\ \#\ es,s'\rangle$

$P \vdash \langle es,s\rangle\ [\rightarrow]\ \langle es',s'\rangle \Longrightarrow P \vdash \langle \mathit{Val}\ v\ \#\ es,s\rangle\ [\rightarrow]\ \langle \mathit{Val}\ v\ \#\ es',s'\rangle$

References

[1] G. Biermann, M. Parkinson, and A. Pitts. Mj: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge, 2003.

[2] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 41–82. Springer-Verlag, 1999.

[3] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 241–269. Springer-Verlag, 1999.

[4] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Comput. Sci.*, 298:583–626, 2003.

[5] H. R. Nielson and F. Nielson. *Semantics with Applications.* Wiley, 1992.

[6] T. Nipkow. Verified bytecode verifiers. In F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lect. Notes in Comp. Sci.*, pages 347–363. Springer-Verlag, 2001.

[7] T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lect. Notes in Comp. Sci.*, pages 259–278. Springer-Verlag, 2003.

[8] T. Nipkow and D. v. Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170, 1998.

[9] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.

[10] D. v. Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.

[11] D. v. Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 119–156. Springer-Verlag, 1999.

[12] N. Schirmer. Java Definite Assignment in in Isabelle/HOL. In S. Eisenbach, G. Leavens, P. Müller, A. Poetzsch-Heffter, and E. Poll, editors, *Formal Techniques for Java-like Programs 2003 (Proceedings).* Chair of Software Engineering, ETH Zürich, 2003. Technical Report 108.

[13] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine — Definition, Verification, Validation.* Springer-Verlag, 2001.

[14] M. Strecker. Formal verification of a Java compiler in Isabelle. In A. Voronkov, editor, *Automated Deduction — CADE-18*, volume 2392 of *Lect. Notes in Comp. Sci.*, pages 63–77. Springer-Verlag, 2002.

[15] D. Syme. Proving Java type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lect. Notes in Comp. Sci.*, pages 83–118. Springer-Verlag, 1999.

[16] M. Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents.* PhD thesis, Institut für Informatik, Technische Universität München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.

[17] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

[18] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.