

A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler

GERWIN KLEIN

National ICT Australia, Sydney

and

TOBIAS NIPKOW

Technische Universität München

We introduce Jinja, a Java-like programming language with a formal semantics designed to exhibit core features of the Java language architecture. Jinja is a compromise between realism of the language and tractability and clarity of the formal semantics. The following aspects are formalised: a big and a small step operational semantics for Jinja and a proof of their equivalence; a type system and a definite initialisation analysis; a type safety proof of the small step semantics; a virtual machine (JVM), its operational semantics and its type system; a type safety proof for the JVM; a bytecode verifier, i.e. data flow analyser for the JVM; a correctness proof of the bytecode verifier w.r.t. the type system; a compiler and a proof that it preserves semantics and well-typedness.

The emphasis of this work is not on particular language features but on providing a *unified* model of the source language, the virtual machine and the compiler. The whole development has been carried out in the theorem prover Isabelle/HOL.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.4 [**Programming Languages**]: Processors—*Compilers*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Object-oriented constructs*

General Terms: Languages, Verification

Additional Key Words and Phrases: Java, operational semantics, theorem proving

1. INTRODUCTION

There is a large body of literature on formal models of Java-like languages, the virtual machine and bytecode verification, and compilation. However, each of these models is designed to treat certain features of Java in depth, while ignoring other features completely (see below for one exception). The main result of our work is the first unified model of a Java-like source language, virtual machine, and compiler, with the following salient features: it is based on operational semantics, it is executable, it fits into the confines of a journal article, and all proofs are machine-checked (in Isabelle/HOL [Nipkow et al. 2002]). This is considerably more than the sum of its parts. A *tractable* formal model is the result of a careful balance of

This work was done while Nipkow was on sabbatical at NICTA where he was supported by the NICTA Formal Methods program and the DFG.

Authors' addresses: Gerwin Klein, National ICT Australia, University of New South Wales, Sydney NSW 2052, Australia. www.cse.unsw.edu.au/~kleing. Tobias Nipkow, Institut für Informatik, Technische Universität München, Boltzmannstraße 3, 85748 Garching, Germany, www.in.tum.de/~nipkow.

features on each level and of the interaction between the different levels (here: the source and target language). That is, integrating different formal models is just as much of a challenge as designing the individual models.

In addition to the unified model, there are also a number of specific advances of the state of the art: a big and small step semantics together with an equivalence proof (previously only one or the other had been used), a formalisation of the “definite assignment” analysis of local variables, and a compiler correctness proof covering exceptions. We discuss these issues in detail in the respective sections.

It must be emphasised that our work is complementary to the detailed analysis of particular language features. In the end, a synthesis of our unified (but in many places simplified) model and the more detailed (but specialised) models should emerge. And because of the unavoidable size of the resulting model, we believe that some machine-checked analysis, ideally with a theorem prover, is necessary to guarantee the overall consistency of the model—in particular when faced with changes. With each change to a model of that size, a purely textual definition becomes less and less reliable: in the absence of mechanical consistency checks, changes in one part of the model will sooner or later have unexpected side effects in other parts. This is where formal verification and the ability to re-run proofs really pays off. But even without formal proofs, mere type checking is already tremendously beneficial. Hence this paper should also be viewed as an attempt to combine the rigour of a formal machine-checked meta-language with standard (largely mathematical) notation. More on this issue in §1.2.

The paper is subdivided into 4 parts: the source language (§2), the virtual machine (§3), the bytecode verifier (§4), and the compiler (§5).

We discuss related work separately at the end of each section because most papers deal with a specific language layer. The exception is the book by Stärk et al. [2001], who treat almost all of Java and the virtual machine. This is a very impressive piece of work, but quite different from ours: it is based on Abstract State Machines rather than standard operational semantics, and the proofs are not machine-checked. It should be noted that the literature on formal models of Java and the JVM is already so large that it gave rise to three survey-like publications [Alves-Foss 1999; Hartel and Moreau 2001; Nipkow 2003a]. Hence our discussion of related work is necessarily restricted to those papers with a very direct connection to ours.

As a final word of warning we must emphasize that the core of the paper is intentionally detailed and technical: its very aim is to demonstrate the state of the art in machine-checked language definitions.

1.1 Basic Notation — The Meta-Language

Our meta-language HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

Types The basic types of truth values, natural numbers and integers are called *bool*, *nat*, and *int*. The space of total functions is denoted by \Rightarrow . Type variables are written *'a*, *'b*, etc. The notation $t::\tau$ means that HOL term t has HOL type τ .

Pairs come with the two projection functions $fst :: 'a \times 'b \Rightarrow 'a$ and $snd :: 'a \times 'b \Rightarrow 'b$. We identify tuples with pairs nested to the right: (a, b, c) is identical to $(a, (b, c))$ and $'a \times 'b \times 'c$ is identical to $'a \times ('b \times 'c)$.

Sets (type *'a set*) follow the usual mathematical convention. Intervals are written as follows: $\{m..<n\}$ means $\{i \mid m \leq i < n\}$ and $\{m..n\}$ means $\{i \mid m \leq i \leq n\}$. If m is 0, it can be dropped.

Lists (type *'a list*) come with the empty list $[]$, the infix constructor \cdot , the infix $@$ that appends two lists, and the conversion function *set* from lists to sets. Variable names ending in “s” usually stand for lists, $|xs|$ is the length of xs , and $xs_{[n]}$, where $n::nat$, is the n th-element of xs (starting with 0). The notation $[i..<j]$ with $i::nat$ and $j::nat$ stands for the list $[i, \dots, j-1]$. *distinct xs* means that the elements of xs are all distinct. The standard functions *map* and *filter* are also available.

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

adjoins a new element *None* to a type *'a*. For succinctness we write $[a]$ instead of *Some a*. The underspecified inverse *the* of *Some* satisfies *the* $[x] = x$.

Function update is written $f(x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$.

Partial functions are modelled as functions of type $'a \Rightarrow 'b \text{ option}$, where *None* represents undefinedness and $f x = [y]$ means x is mapped to y . We define $dom\ m \equiv \{a \mid m\ a \neq None\}$. Instead of $'a \Rightarrow 'b \text{ option}$ we write $'a \multimap 'b$, call such functions **maps**, and abbreviate $f(x := [y])$ to $f(x \mapsto y)$. The latter notation extends to lists: $f([x_1, \dots, x_m] \mapsto [y_1, \dots, y_n])$ means $f(x_1 \mapsto y_1) \dots (x_i \mapsto y_i)$, where i is the minimum of m and n . The notation works for arbitrary list expressions on both sides of \mapsto , not just enumerations. Multiple updates like $f(x \mapsto y)(xs \mapsto ys)$ can be written as $f(x \mapsto y, xs \mapsto ys)$. The map $\lambda x. None$ is written *empty*, and *empty(...)*, where ... are updates, abbreviates to $[...]$. For example, $empty(x \mapsto y, xs \mapsto ys)$ becomes $[x \mapsto y, xs \mapsto ys]$.

Overwriting map m_1 with m_2 is written $m_1 ++ m_2$ and means $\lambda x. \text{case } m_2\ x\ \text{of } None \Rightarrow m_1\ x \mid [y] \Rightarrow [y]$.

Function *map-of* turns an association list, i.e. list of pairs, into a map:

map-of $[] = empty$
map-of $(p \cdot ps) = \text{map-of } ps(fst\ p \mapsto snd\ p)$

Note that $[A_1; \dots; A_n] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\dots \Longrightarrow (A_n \Longrightarrow A) \dots)$. In lemmas we often write “If A_1 and ... and A_n then A ” instead. Displayed implications are frequently printed as inference rules:

$$\frac{A_1 \quad A_2 \quad A_3}{A}$$

1.2 Presentation Issues

How are the formulae you see related to the formal Isabelle text? Our motto is

What you see is what we proved!

In many papers this is not the case. For example, the main definition by Nipkow [1991] and the main theorem, Theorem 1, by Flatt et al. [1999] are blatantly wrong (the latter misses the case that e might diverge). In order to avoid this problem, Isabelle theories can be augmented with \LaTeX text which may contain references to Isabelle theorems (by name — see chapter 4 of the book by Nipkow et al. [2002]). When this \LaTeX text is processed by Isabelle, it expands these references into the \LaTeX text for the proposition of the theorem. Using this mechanism, the text for

most of the definitions and theorems in this paper is automatically generated. We emphasize that the \LaTeX presentation is quite close to the original Isabelle theories viewed with the help of the Proof General interface [Aspinall 2000] because the latter supports mathematical symbols. Only a few niceties like inference rules are \LaTeX specific.

It is conceivable to go one step further and even generate proof text automatically from the theories. However, although most of our proofs are in a quasi-readable form of structured stylized mathematics [Wenzel 2002; Nipkow 2003b], it appears beyond the state of the art to turn these into concise journal-style proofs automatically.

2. JINJA

Our dialect of Java is called **Jinja** (because Jinja is not Java). Although Jinja is a typed language, we begin its description with the operational semantics which is independent of the type system. First we introduce a big step or evaluation semantics (§2.2), then a small step or reduction semantics (§2.3). The big step semantics will be used in the compiler proof, the small step semantics in the type safety proof. Both semantics are *defensive*: rules only apply if everything “fits together”. For example, variable lookup requires that the variable has been initialised. This is a prerequisite for the type safety proof, which shows that reduction of well-typed expressions does not get stuck. In §2.4 certain minimal well-formedness conditions for programs are defined. They imply (§2.5) that the two semantics are equivalent for terminating executions. Then we introduce a type system (§2.6) and a “definite assignment” analysis (§2.7) for expressions and complete the set of well-formedness conditions for Jinja programs (§2.8). Finally we show type safety (§2.9).

2.1 Abstract Syntax

2.1.1 Names. In the sequel we use the following (HOL) variable conventions: V is a (Jinja) 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.

For readability only we have introduced three (HOL) types for Jinja identifiers: *cname* (class names), *vname* (variable names), and *mname* (method names). All three are merely synonyms for type *string*.

2.1.2 Values and Expressions. A Jinja **value** can be

- a boolean *Bool* b , where $b :: \text{bool}$, or
- an integer *Intg* i , where $i :: \text{int}$, or
- a reference *Addr* a , where $a :: \text{addr}$, 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*. To make things executable, type *addr* is actually a synonym for type *nat* (as opposed to leaving it unspecified).

The following **expressions** (of HOL type *expr*) are supported by Jinja:

- creation of new object: `new C`
- casting: `Cast C e`
- literal value: `Val v`
- binary operation: $e_1 \ll bop \gg e_2$ (where *bop* is one of + or =)
- variable access `Var V` and variable assignment `V := e`
- field access $e.F\{D\}$ and field assignment $e_1.F\{D\} := e_2$
(where *D* is the class where *F* is declared)
- method call: $e.M(es)$
- block with locally declared variable: $\{V:T; e\}$
- sequential composition: $e_1; e_2$
- conditional: `if (e) e1 else e2` (do not confuse with HOL's *if b then x else y*)
- while loop: `while (e) e'`
- exception throwing `throw e` and catching `try e1 catch (C V) e2`

The constructors `Val` and `Var` are needed in our meta-language to disambiguate the syntax. There is no return statement because everything is an expression and returns a value.

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 §2.6.

To ease notation we introduce some abbreviations:

$$\begin{array}{lll}
 \text{true} & \equiv & \text{Val}(\text{Bool True}) \quad \text{false} \equiv \text{Val}(\text{Bool False}) \\
 \text{addr } a & \equiv & \text{Val}(\text{Addr } a) \quad \text{null} \equiv \text{Val Null} \\
 \text{unit} & \equiv & \text{Val Unit}
 \end{array}$$

Jinja supports only the two binary operators = and + to keep things simple. Their evaluation is defined via a function *binop* that takes an operator and two values and returns an optional value (to deal with type mismatches):

$$\begin{array}{ll}
 \text{binop } (=, v_1, v_2) & = \text{[Bool } (v_1 = v_2)\text{]} \\
 \text{binop } (+, \text{Intg } i_1, \text{Intg } i_2) & = \text{[Intg } (i_1 + i_2)\text{]} \\
 \text{binop } (bop, v_1, v_2) & = \text{None}
 \end{array}$$

Addition only yields a value if both arguments are integers. We could also insist on similar compatibility checks for the equality test, but that leads to excessive case distinctions that we want to avoid for reasons of presentation.

2.1.3 Programs. The abstract syntax of programs is given by the type definitions in Fig. 1, where *ty* is the HOL type of Jinja types. A program is a list of class declarations. A **class declaration** consists of the name of the class and the class itself. A **class** consists of the name of its direct superclass, a list of field declarations and a list of method declarations. A **field declaration** is a pair of a field name and its type. A **method declaration** consists of the method name, the parameter types, the result type, and the method body.

The only unusual thing here is the parameterisation of these types by '*m*', the type of the method body. The reason is that we want to use the program structure not just for Jinja but also for virtual machine programs. All we need to do is to slot

types	$'m \text{ prog}$	$=$	$'m \text{ cdecl list}$
	$'m \text{ cdecl}$	$=$	$cname \times 'm \text{ class}$
	$'m \text{ class}$	$=$	$cname \times fdecl \text{ list} \times 'm \text{ mdecl list}$
	$fdecl$	$=$	$vname \times ty$
	$'m \text{ mdecl}$	$=$	$mname \times ty \text{ list} \times ty \times 'm$
	$J\text{-mb}$	$=$	$vname \text{ list} \times expr$
	$J\text{-prog}$	$=$	$J\text{-mb prog}$

Fig. 1. Abstract program syntax

in the right kind of method body: a Jinja method body $J\text{-mb}$ is a pair of the formal parameter names and the expression, a Jinja program $J\text{-prog}$ a program with Jinja method bodies. Note that parameter names cannot be part of the generic syntax because there are no parameter names on the virtual machine level.

This generic program syntax is an important abstraction that will also come in handy during compilation. More concrete representations, e.g. concrete source language syntax or Java’s class file format, are orthogonal to our work and can be added separately.

2.1.4 Extracting Declaration Information. Most of the time the exact representation of programs is irrelevant because we work in terms of a few functions and predicates for analysing and accessing the declarations in a program:

- **class** $P \ C$ is the class associated with C in P .
- **is-class** $P \ C$ means class C is defined in P .
- $P \vdash D \preceq^* C$ means D is a **subclass** of C . The relation is transitive and reflexive.
- $P \vdash C \text{ sees } M:Ts \rightarrow T = mb \text{ in } D$ means that in P , scanning the class hierarchy upwards starting from C , a method M is visible in class D (taking overriding into account). Its argument types are Ts (a type list), result type is T , and body is mb . If P is a Jinja program, mb is a pair (pns, e) of formal parameter list pns and expression e .
- $P \vdash C \text{ sees } F:T \text{ in } D$ means that in P from class C a field F of type T is visible in class D .
- $P \vdash C \text{ has } F:T \text{ in } D$ means that in P a (not necessarily proper) superclass D of C has a field F of type T .

Before we show the definition of these predicates we give an example (in an imaginary concrete syntax) that should clarify the concepts:

```
class B extends A {field F:TB
                    method M:TBs->T1 = mB}
class C extends B {field F:TC
                    method M:TCs->T2 = mC}
```

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

Analogously we have $P \vdash B \text{ sees } M: TBs \rightarrow T_1 = mB \text{ in } B$ and $P \vdash C \text{ sees } M: TCs \rightarrow T_2 = mC \text{ in } C$, but not $P \vdash C \text{ sees } M: TBs \rightarrow T_1 = mB \text{ in } B$. The second

declaration of M overrides the first, no matter what the type of M in the two declarations is.

This is method overriding in its purest form but 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 formalised overloading elsewhere [Oheimb and Nipkow 1999] but have not included it in Jinja. It complicates matters without adding a significant new aspect: one has to annotate method calls with the static type of the actual parameter list, just as in field access. This mixture of static type (for the parameter list) and dynamic type (of the object at run time) can make programs with overloading hard to understand.

We will now show how the above functions are defined. The first two are trivial: $class \equiv map-of$ and $is-class P C \equiv class P C \neq None$.

For the remaining functions it is important to note that we do not differentiate between user defined classes and system classes. That is, any proper program will need to contain a class $Object$. Since any class has a superclass entry, so has $Object$. Therefore class $Object$ is treated specially: traversal of the class hierarchy stops there and the dummy superclass entry is ignored.

The subclass relation \preceq^* is defined as the reflexive transitive closure of the direct subclass relation \prec^1 defined by

$$\frac{class P C = \lfloor(D, fs, ms)\rfloor \quad C \neq Object}{P \vdash C \prec^1 D}$$

The information about fields in a class hierarchy is collected by a predicate $P \vdash C \text{ has-fields } FDTs$ where $FDTs :: ((vname \times cname) \times ty) \text{ list}$. That is, each tuple $((F, D), T)$ in $FDTs$ represents a field declaration $F:T$ in a superclass D of C , and the list follows the class hierarchy, i.e. the fields of C itself come first in the list. The predicate is defined inductively:

$$\frac{class P C = \lfloor(D, fs, ms)\rfloor \quad C \neq Object \quad P \vdash D \text{ has-fields } FDTs}{P \vdash C \text{ has-fields } map(\lambda(F, T). ((F, C), T)) fs @ FDTs}$$

$$\frac{class P Object = \lfloor(D, fs, ms)\rfloor}{P \vdash Object \text{ has-fields } map(\lambda(F, T). ((F, Object), T)) fs}$$

At the moment we do not rule out class $Object$ having fields. In our example above, assuming that A is in fact $Object$, and assuming that $Object$ does not have fields, we obtain $P \vdash C \text{ has-fields } [((F, C), TC), ((F, B), TB)]$.

From the $has-fields$ relation we can define has and $sees$ directly:

$$P \vdash C \text{ has } F:T \text{ in } D \equiv \exists FDTs. P \vdash C \text{ has-fields } FDTs \wedge map-of FDTs (F, D) = \lfloor T \rfloor$$

$$P \vdash C \text{ sees } F:T \text{ in } D \equiv$$

$$\exists FDTs. P \vdash C \text{ has-fields } FDTs \wedge$$

$$map-of (map(\lambda((F, D), T). (F, D, T)) FDTs) F = \lfloor(D, T)\rfloor$$

The relation $sees$ for methods can be defined analogously via a relation that traverses the class hierarchy and collects all method declaration information. We omit the details.

2.2 Big Step Semantics

2.2.1 State. The type of states during expression evaluation is defined in Fig. 2. A **state** is a pair of a **heap** and a **store** (*locals*). A store is a map from variable

$$\begin{array}{llll}
\mathbf{types} & state & = & heap \times locals & obj & = & cname \times fields \\
& locals & = & vname \rightarrow val & fields & = & vname \times cname \rightarrow val \\
& heap & = & addr \rightarrow obj & & &
\end{array}$$

Fig. 2. The type of Jinja program states

names to values. A heap is a 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, all of them visible.

The naming convention is that h is a heap, l is a store (the local variables), and s a state. The projection functions hp and lcl are synonyms for fst and snd .

When a new object is allocated on the heap, its fields are initialised with the default value determined by their type:

$$\begin{array}{l}
init\text{-}fields :: ((vname \times cname) \times ty) list \Rightarrow fields \\
init\text{-}fields \equiv map\text{-}of \circ map (\lambda(F, T). (F, default\text{-}val T))
\end{array}$$

The definition of the default value is irrelevant for our purposes. It suffices to know that the default value is of the right type and that for references (types *Class* and *NT*) it is *Null*.

2.2.2 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.*

We will discuss the evaluation rules in an incremental fashion: first normal evaluation only, exceptional behaviour afterwards.

2.2.3 Normal Evaluation. Normal evaluation means that we are defining an exception-free language. In particular, all final expressions will be values. The complete set of rules is shown in Fig. 3 and we discuss them in turn.

new C first allocates a new address: function *new-Addr* (we omit its definition) returns a “new” address, i.e. $new\text{-}Addr\ h = [a]$ implies $h\ a = None$. Then predicate *has-fields* (§2.1.4) computes the list of all field declarations in and above C , and *init-fields* (§2.2.1) creates the default field table.

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.

The rules for **Val**, **Var** and assignment are self-explanatory.

Field access $e.F\{D\}$ 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.

$$\begin{array}{c}
 \frac{\text{new-Addr } h = [a] \quad P \vdash C \text{ has-fields FDTs} \quad h' = h(a \mapsto (C, \text{init-fields FDTs}))}{P \vdash \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{addr } a, (h', l) \rangle} \\
 \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle \quad h a = [(D, fs)] \quad P \vdash D \preceq^* C}{P \vdash \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle} \\
 \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle}{P \vdash \langle \text{Cast } C e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle} \\
 \\
 P \vdash \langle \text{Val } v, s \rangle \Rightarrow \langle \text{Val } v, s \rangle \\
 \\
 \frac{l V = [v]}{P \vdash \langle \text{Var } V, (h, l) \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle} \\
 \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle \quad l' = l(V \mapsto v)}{P \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \text{unit}, (h, l') \rangle} \\
 \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle \quad h a = [(C, fs)] \quad fs(F, D) = [v]}{P \vdash \langle e.F\{D\}, s_0 \rangle \Rightarrow \langle \text{Val } v, (h, l) \rangle} \\
 \\
 \frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle \quad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, (h_2, l_2) \rangle}{h_2 a = [(C, fs)] \quad fs' = fs((F, D) \mapsto v) \quad h_2' = h_2(a \mapsto (C, fs'))} \\
 \frac{P \vdash \langle e_1.F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{unit}, (h_2', l_2) \rangle}{} \\
 \\
 \frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle \quad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v_2, s_2 \rangle \quad \text{binop}(bop, v_1, v_2) = [v]}{P \vdash \langle e_1 \ll bop \gg e_2, s_0 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle} \\
 \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle \quad P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, (h_2, l_2) \rangle}{h_2 a = [(C, fs)] \quad P \vdash C \text{ sees } M: Ts \rightarrow T = (pns, \text{body}) \text{ in } D} \\
 \frac{|vs| = |pns| \quad l_2' = [this \mapsto \text{Addr } a, pns [\rightarrow] vs] \quad P \vdash \langle \text{body}, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle}{P \vdash \langle e.M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle} \\
 \\
 \frac{P \vdash \langle e_0, (h_0, l_0(V := \text{None})) \rangle \Rightarrow \langle e_1, (h_1, l_1) \rangle}{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} \\
 \\
 \frac{P \vdash \langle e_0, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle \quad P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e_2, s_2 \rangle}{P \vdash \langle e_0; e_1, s_0 \rangle \Rightarrow \langle e_2, s_2 \rangle} \\
 \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle \quad P \vdash \langle e_1, s_1 \rangle \Rightarrow \langle e', s_2 \rangle}{P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle} \\
 \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle \quad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle e', s_2 \rangle}{P \vdash \langle \text{if } (e) e_1 \text{ else } e_2, s_0 \rangle \Rightarrow \langle e', s_2 \rangle} \\
 \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{false}, s_1 \rangle}{P \vdash \langle \text{while } (e) c, s_0 \rangle \Rightarrow \langle \text{unit}, s_1 \rangle} \\
 \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{true}, s_1 \rangle \quad P \vdash \langle c, s_1 \rangle \Rightarrow \langle \text{Val } v_1, s_2 \rangle \quad P \vdash \langle \text{while } (e) c, s_2 \rangle \Rightarrow \langle e_3, s_3 \rangle}{P \vdash \langle \text{while } (e) c, s_0 \rangle \Rightarrow \langle e_3, s_3 \rangle} \\
 \\
 \frac{P \vdash \langle [], s \rangle [\Rightarrow] \langle [], s \rangle \quad P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{Val } v, s_1 \rangle \quad P \vdash \langle es, s_1 \rangle [\Rightarrow] \langle es', s_2 \rangle}{P \vdash \langle e \cdot es, s_0 \rangle [\Rightarrow] \langle \text{Val } v \cdot es', s_2 \rangle}
 \end{array}$$

Fig. 3. Normal evaluation of expressions

Field assignment $e_1.F\{D\} := e_2$ 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 *unit* (just like assignment to local variables).

Why does assignment not evaluate to the value of the rhs, like in Java? Because then the conditional `if (e) V1 := e1 else V2 := e2` would not evaluate to *unit* but to the value of e_1 or e_2 . Thus the type system (to be defined later) would require e_1 and e_2 to have compatible types, which in many cases they wouldn't, thus forcing the programmer to write something like `if (e) (V1 := e1; unit) else (V2 := e2; unit)`.

Binary operations are evaluated from left to right (for *binop* see §2.1.2).

The lengthiest rule is the one for method call. Its reading is easy: evaluate e to an address a and the parameter list ps to a list of values vs ¹, 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 , and evaluate the body in a store that maps *this* to *Addr a* and the formal parameter names to the actual parameter values (having made sure that vs and pns have the same length). The final store is the one obtained from the evaluation of the parameters — the one obtained from the evaluation of $body$ is discarded. This rule reflects a well-formedness condition imposed later on: there are no global variables in Jinja (just as in Java, but contrary to, say, C++), i.e. a method body should only refer to *this* and its parameters.

In Jinja, blocks with local variables, sequential composition, conditional and while-loop are expressions too, in contrast to Java, where they are commands and do not return a value. 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. Sequential composition discards the value of the first expression. Similarly, while-loops discard the value of their body and, upon termination, return *unit*.

The rules for $[\Rightarrow]$, the evaluation of expression lists (needed for method call), define that lists are evaluated from left to right. This concludes the complete semantics of the exception-free fragment of Jinja.

2.2.4 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 $(\text{Var } V, (h,l))$ only succeeds if $V \in \text{dom } l$. Later on (§2.7), a static analysis (“definite assignment”) will identify expressions where $V \in \text{dom } l$ always holds. Thus we do not need a rule for the situation where $V \notin \text{dom } l$. 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 `throw(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 the system exception classes:

$\text{sys-xcpts} \equiv \{\text{NullPointer}, \text{ClassCast}, \text{OutOfMemory}\}$

¹ $[\Rightarrow]$ is evaluation extended to lists of expressions. Saying that the result is of the form `map Val vs` is a declarative way of ensuring that it is a list of values and of obtaining the actual value list vs (as opposed to an expression list).

$$\begin{array}{c}
 \frac{\text{new-Addr } h = \text{None}}{P \vdash \langle \text{new } C, (h, l) \rangle \Rightarrow \langle \text{THROW } \text{OutOfMemory}, (h, l) \rangle} \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, (h, l) \rangle \quad h \ a = \llbracket (D, fs) \rrbracket \quad \neg P \vdash D \preceq^* C}{P \vdash \langle \text{Cast } C \ e, s_0 \rangle \Rightarrow \langle \text{THROW } \text{ClassCast}, (h, l) \rangle} \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle}{P \vdash \langle e.F\{D\}, s_0 \rangle \Rightarrow \langle \text{THROW } \text{NullPointer}, s_1 \rangle} \\
 \frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \quad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \text{Val } v, s_2 \rangle}{P \vdash \langle e_1.F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \text{THROW } \text{NullPointer}, s_2 \rangle} \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle \quad P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \text{map Val } vs, s_2 \rangle}{P \vdash \langle e.M(ps), s_0 \rangle \Rightarrow \langle \text{THROW } \text{NullPointer}, s_2 \rangle} \\
 \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{addr } a, s_1 \rangle}{P \vdash \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{Throw } a, s_1 \rangle} \quad \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \text{null}, s_1 \rangle}{P \vdash \langle \text{throw } e, s_0 \rangle \Rightarrow \langle \text{THROW } \text{NullPointer}, s_1 \rangle} \\
 \frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle}{P \vdash \langle \text{try } e_1 \text{ catch } (C \ V) \ e_2, s_0 \rangle \Rightarrow \langle \text{Val } v_1, s_1 \rangle} \\
 \frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, l_1) \rangle \quad h_1 \ a = \llbracket (D, fs) \rrbracket \quad P \vdash D \preceq^* C \quad P \vdash \langle e_2, (h_1, l_1(V \mapsto \text{Addr } a)) \rangle \Rightarrow \langle e_2', (h_2, l_2) \rangle}{P \vdash \langle \text{try } e_1 \text{ catch } (C \ V) \ e_2, s_0 \rangle \Rightarrow \langle e_2', (h_2, l_2(V := l_1 \ V)) \rangle} \\
 \frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, l_1) \rangle \quad h_1 \ a = \llbracket (D, fs) \rrbracket \quad \neg P \vdash D \preceq^* C}{P \vdash \langle \text{try } e_1 \text{ catch } (C \ V) \ e_2, s_0 \rangle \Rightarrow \langle \text{Throw } a, (h_1, l_1) \rangle}
 \end{array}$$

Fig. 4. Throwing and catching exceptions

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* :: *cname* \Rightarrow *addr* whose precise definition is not important. To ease notation we introduce some abbreviations:

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

2.2.5 Exceptional Evaluation. The basic rules for throwing and catching exceptions are shown in Fig. 4. 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, field update, or method call is null. The **throw** construct may throw any expression of class type, which is a simplification of Java’s exceptions. Throwing *null* leads to a *NullPointer* exception.

Note that we have maintained Java’s eager evaluation scheme of evaluating all subterms before throwing any system exception. This permits the simple compilation strategy where the values of the subterms are first put on the stack unchecked and the check is performed at the end by the machine instruction, e.g. field access, accessing the object reference in question.

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

$$\begin{array}{c}
\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle}{P \vdash \langle \mathbf{Cast} \ C \ e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle} \quad \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle}{P \vdash \langle V := e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle} \\
\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle}{P \vdash \langle e.F\{D\}, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle} \quad \frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle}{P \vdash \langle e_1.F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle} \\
\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \mathbf{Val} \ v, s_1 \rangle \quad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_2 \rangle}{P \vdash \langle e_1.F\{D\} := e_2, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_2 \rangle} \\
\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e, s_1 \rangle}{P \vdash \langle e_1 \ll \mathit{bop} \gg e_2, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e, s_1 \rangle} \\
\frac{P \vdash \langle e_1, s_0 \rangle \Rightarrow \langle \mathbf{Val} \ v_1, s_1 \rangle \quad P \vdash \langle e_2, s_1 \rangle \Rightarrow \langle \mathbf{throw} \ e, s_2 \rangle}{P \vdash \langle e_1 \ll \mathit{bop} \gg e_2, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e, s_2 \rangle} \\
\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle}{P \vdash \langle e.M(ps), s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle} \\
\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{Val} \ v, s_1 \rangle \quad P \vdash \langle es, s_1 \rangle [\Rightarrow] \langle \mathit{map} \ \mathbf{Val} \ vs \ @ \ (\mathbf{throw} \ ex \cdot es'), s_2 \rangle}{P \vdash \langle e.M(es), s_0 \rangle \Rightarrow \langle \mathbf{throw} \ ex, s_2 \rangle} \\
\frac{P \vdash \langle e_0, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e, s_1 \rangle}{P \vdash \langle e_0; e_1, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e, s_1 \rangle} \quad \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle}{P \vdash \langle \mathbf{if} \ (e) \ e_1 \ \mathbf{else} \ e_2, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle} \\
\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle \quad P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathit{true}, s_1 \rangle \quad P \vdash \langle c, s_1 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_2 \rangle}{P \vdash \langle \mathbf{while} \ (e) \ c, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle} \quad \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle}{P \vdash \langle \mathbf{while} \ (e) \ c, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_2 \rangle} \\
\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle}{P \vdash \langle \mathbf{throw} \ e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle} \quad \frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \mathbf{throw} \ e', s_1 \rangle}{P \vdash \langle e \cdot es, s_0 \rangle [\Rightarrow] \langle \mathbf{throw} \ e' \cdot es, s_1 \rangle}
\end{array}$$

Fig. 5. Exception propagation

to *Addr* a and e_2 is evaluated; otherwise **Throw** a is the result of the evaluation.

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 and shown in Fig. 5. This concludes the exposition of the evaluation rules.

A compact representation of the 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

$$\frac{P \vdash \langle e, s \rangle \Rightarrow \langle \mathbf{throw} \ e', s' \rangle}{P \vdash \langle C_x[e], s \rangle \Rightarrow \langle \mathbf{throw} \ e', s' \rangle}$$

We prefer not to formalize these additional notions and stay within a fixed basic framework of ordinary expressions.

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

$$\mathit{final} \ e \equiv (\exists v. e = \mathbf{Val} \ v) \vee (\exists a. e = \mathbf{Throw} \ a)$$

LEMMA 2.1. 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 2.1. 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 $P \vdash \langle es, s \rangle [\Rightarrow] \langle es', s' \rangle \Longrightarrow \forall e' \in \text{set } es'. \text{ final } e'$. 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:

$$\text{finals } es \equiv (\exists vs. es = \text{map Val } vs) \vee (\exists vs a es'. es = \text{map Val } vs @ (\text{Throw } a \cdot es'))$$

and Lemma 2.1 for lists is 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 2.2. If *final* e then $P \vdash \langle e, s \rangle \Rightarrow \langle e, s \rangle$.

Of course an analogous lemma holds for expression lists, but we have chosen not to show it. We will follow this practice whenever the list version of a theorem is obvious. In fact, one could dispose of expressions lists altogether by restricting Jinja methods to a single parameter. However, this is precisely the kind of simplification we do not want to make, because it would give the wrong impression that including expression lists could be a significant burden.

2.3 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 §2.9 needs a finer grained semantics. Otherwise we cannot prove that type-correct expressions do not get stuck during evaluation because the big step semantics does not distinguish between divergence (of nonterminating expressions) and deadlock (of ill-typed expressions). 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 reduction 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 $\langle e_1, s_1 \rangle \rightarrow \langle e_2, s_2 \rangle \dots \rightarrow \langle e_n, s_n \rangle$ to reduce an expression completely.

As for the big step semantics we can define normal and exceptional reductions separately. We begin with normal reductions. The 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.

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

$$\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle c \dots e \dots, s \rangle \rightarrow \langle c \dots e' \dots, s' \rangle}$$

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 Jinja subexpression reduction are shown in Fig. 6. The initial ones follow the pattern above exactly. For

$$\begin{array}{c}
\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle \text{Cast } C \ e, s \rangle \rightarrow \langle \text{Cast } C \ e', s' \rangle} \quad \frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle V := e, s \rangle \rightarrow \langle V := e', s' \rangle} \\
\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle e.F\{D\}, s \rangle \rightarrow \langle e'.F\{D\}, s' \rangle} \quad \frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle e.F\{D\} := e_2, s \rangle \rightarrow \langle e'.F\{D\} := e_2, s' \rangle} \\
\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle \text{Val } v.F\{D\} := e, s \rangle \rightarrow \langle \text{Val } v.F\{D\} := e', s' \rangle} \quad \frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle e \ll bop \gg e_2, s \rangle \rightarrow \langle e' \ll bop \gg e_2, s' \rangle} \\
\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle \text{Val } v_1 \ll bop \gg e, s \rangle \rightarrow \langle \text{Val } v_1 \ll bop \gg e', s' \rangle} \\
\frac{P \vdash \langle e, (h, l(V := \text{None})) \rangle \rightarrow \langle e', (h', l') \rangle \quad l' V = \text{None} \quad \neg \text{assigned } V \ e}{P \vdash \langle \{V:T; e\}, (h, l) \rangle \rightarrow \langle \{V:T; e'\}, (h', l'(V := l V)) \rangle} \\
\frac{P \vdash \langle e, (h, l(V := \text{None})) \rangle \rightarrow \langle e', (h', l') \rangle \quad l' V = [v] \quad \neg \text{assigned } V \ e}{P \vdash \langle \{V:T; e\}, (h, l) \rangle \rightarrow \langle \{V:T; V := \text{Val } v; e'\}, (h', l'(V := l V)) \rangle} \\
\frac{P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow \langle e', (h', l') \rangle \quad l' V = [v']}{P \vdash \langle \{V:T; V := \text{Val } v; e\}, (h, l) \rangle \rightarrow \langle \{V:T; V := \text{Val } v'; e'\}, (h', l'(V := l V)) \rangle} \\
\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle e.M(es), s \rangle \rightarrow \langle e'.M(es), s' \rangle} \quad \frac{P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle}{P \vdash \langle \text{Val } v.M(es), s \rangle \rightarrow \langle \text{Val } v.M(es'), s' \rangle} \\
\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle e; e_2, s \rangle \rightarrow \langle e'; e_2, s' \rangle} \quad \frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle \text{if } (e) \ e_1 \ \text{else } e_2, s \rangle \rightarrow \langle \text{if } (e') \ e_1 \ \text{else } e_2, s' \rangle} \\
\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle e \cdot es, s \rangle [\rightarrow] \langle e' \cdot es, s' \rangle} \quad \frac{P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle}{P \vdash \langle \text{Val } v \cdot es, s \rangle [\rightarrow] \langle \text{Val } v \cdot es', s' \rangle}
\end{array}$$

Fig. 6. Subexpression reduction

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. 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 $\{V:T; V := \text{Val } v; e'\}$. The final rule reduces such blocks. This additional rule is necessary because $\{V:T; V := \text{Val } v; e\}$ must not be reduced as before, by reducing all of $V := \text{Val } v; e$ to e (thus losing the binding for V), but by reducing e directly. To this end we have introduced the predicate

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

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

Note that we cannot treat local variables simply by creating “new” variables because we do not know which other variables exist in the context: $\text{dom } l$ does not contain all of them because variables need not be initialized upon creation.

To reduce a method call, the object expression is reduced until it has become an address, and then the parameters are reduced. The relation $[\rightarrow]$ is the extension of \rightarrow to expression lists, which are reduced from left to right, and each element is reduced until it has become a value.

$$\begin{array}{c}
\frac{\text{new-Addr } h = [a] \quad P \vdash C \text{ has-fields } FDTs}{P \vdash \langle \mathbf{new } C, (h, l) \rangle \rightarrow \langle \text{addr } a, (h(a \mapsto (C, \text{init-fields } FDTs))), l \rangle} \\
\frac{\text{hp } s \ a = [(D, fs)] \quad P \vdash D \preceq^* C}{P \vdash \langle \mathbf{Cast } C (\text{addr } a), s \rangle \rightarrow \langle \text{addr } a, s \rangle} \quad P \vdash \langle \mathbf{Cast } C \ \text{null}, s \rangle \rightarrow \langle \text{null}, s \rangle \\
\frac{\text{lcl } s \ V = [v]}{P \vdash \langle \mathbf{Var } V, s \rangle \rightarrow \langle \mathbf{Val } v, s \rangle} \quad P \vdash \langle V := \mathbf{Val } v, (h, l) \rangle \rightarrow \langle \text{unit}, (h, l(V \mapsto v)) \rangle \\
\frac{\text{binop } (bop, v_1, v_2) = [v]}{P \vdash \langle \mathbf{Val } v_1 \ll bop \gg \mathbf{Val } v_2, s \rangle \rightarrow \langle \mathbf{Val } v, s \rangle} \quad \frac{\text{hp } s \ a = [(C, fs)] \quad fs(F, D) = [v]}{P \vdash \langle \text{addr } a.F\{D\}, s \rangle \rightarrow \langle \mathbf{Val } v, s \rangle} \\
\frac{h \ a = [(C, fs)]}{P \vdash \langle \text{addr } a.F\{D\} := \mathbf{Val } v, (h, l) \rangle \rightarrow \langle \text{unit}, (h(a \mapsto (C, fs((F, D) \mapsto v))), l) \rangle} \\
\frac{\text{hp } s \ a = [(C, fs)] \quad P \vdash C \text{ sees } M: Ts \rightarrow T = (pns, \text{body}) \text{ in } D \quad |vs| = |pns| \quad |Ts| = |pns|}{P \vdash \langle \text{addr } a.M(\text{map } \mathbf{Val } vs), s \rangle \rightarrow \langle \text{blocks } (this \cdot pns, \text{Class } D \cdot Ts, \text{Addr } a \cdot vs, \text{body}), s \rangle} \\
P \vdash \langle \{V:T; V := \mathbf{Val } v; \mathbf{Val } u\}, s \rangle \rightarrow \langle \mathbf{Val } u, s \rangle \quad P \vdash \langle \{V:T; \mathbf{Val } u\}, s \rangle \rightarrow \langle \mathbf{Val } u, s \rangle \\
P \vdash \langle \mathbf{Val } v; e_2, s \rangle \rightarrow \langle e_2, s \rangle \\
P \vdash \langle \mathbf{if } (true) \ e_1 \ \mathbf{else } \ e_2, s \rangle \rightarrow \langle e_1, s \rangle \quad P \vdash \langle \mathbf{if } (false) \ e_1 \ \mathbf{else } \ e_2, s \rangle \rightarrow \langle e_2, s \rangle \\
P \vdash \langle \mathbf{while } (b) \ c, s \rangle \rightarrow \langle \mathbf{if } (b) \ (c; \mathbf{while } (b) \ c) \ \mathbf{else } \ \text{unit}, s \rangle
\end{array}$$

Fig. 7. Expression reductions

2.3.2 *Expression Reduction.* Once the subexpressions are sufficiently reduced, we can reduce the whole expression. The rules are shown in Fig. 7. Most of the rules are fairly intuitive and many resemble their big step counterparts. The only one that deserves some explanation is the one for method invocation. In order to avoid explicit stacks we use local variables to hold the values of the parameters. The required nested block structure is built with the help of the auxiliary function *blocks* of type $vname \ list \times \ ty \ list \times \ val \ list \times \ expr \Rightarrow \ expr$:

$$\begin{aligned}
\text{blocks } (V \cdot Vs, T \cdot Ts, v \cdot vs, e) &= \{V:T; V := \mathbf{Val } v; \text{blocks } (Vs, Ts, vs, e)\} \\
\text{blocks } ([], [], [], e) &= e
\end{aligned}$$

Note that we can only get away with this simple rule for method call because there are no global variables in Java. Otherwise one could unfold a method body that refers to some global variable into a context that declares a local variable of the same name, which would essentially amount to dynamic variable binding.

2.3.3 *Exceptional Reduction.* The rules for exception throwing are shown in Fig. 8. System exceptions are thrown almost exactly like in the big step semantics. Expression **throw** e is reduced by reducing e as long as possible and throwing *NullPointerException* if necessary. And this is how **try** e **catch** $(C \ V) \ e_2$ is reduced: First we must reduce e . 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 shown in Fig. 9.

It should be noted that $\{V:T; \mathbf{throw } e\}$ 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

$$\begin{array}{c}
\frac{\text{new-Addr } h = \text{None}}{P \vdash \langle \text{new } C, (h, l) \rangle \rightarrow \langle \text{THROW } \text{OutOfMemory}, (h, l) \rangle} \\
\frac{hp \ s \ a = \lfloor (D, fs) \rfloor \quad \neg P \vdash D \preceq^* C}{P \vdash \langle \text{Cast } C \ (\text{addr } a), s \rangle \rightarrow \langle \text{THROW } \text{ClassCast}, s \rangle} \\
P \vdash \langle \text{null.F}\{D\}, s \rangle \rightarrow \langle \text{THROW } \text{NullPointer}, s \rangle \\
P \vdash \langle \text{null.F}\{D\} := \text{Val } v, s \rangle \rightarrow \langle \text{THROW } \text{NullPointer}, s \rangle \\
P \vdash \langle \text{null.M}(\text{map Val } vs), s \rangle \rightarrow \langle \text{THROW } \text{NullPointer}, s \rangle \\
\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle \text{throw } e, s \rangle \rightarrow \langle \text{throw } e', s' \rangle} \quad P \vdash \langle \text{throw null}, s \rangle \rightarrow \langle \text{THROW } \text{NullPointer}, s \rangle \\
\frac{P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle}{P \vdash \langle \text{try } e \ \text{catch } (C \ V) \ e_2, s \rangle \rightarrow \langle \text{try } e' \ \text{catch } (C \ V) \ e_2, s' \rangle} \\
P \vdash \langle \text{try Val } v \ \text{catch } (C \ V) \ e_2, s \rangle \rightarrow \langle \text{Val } v, s \rangle \\
\frac{hp \ s \ a = \lfloor (D, fs) \rfloor \quad P \vdash D \preceq^* C}{P \vdash \langle \text{try Throw } a \ \text{catch } (C \ V) \ e_2, s \rangle \rightarrow \langle \{V: \text{Class } C; V := \text{addr } a; e_2\}, s \rangle} \\
\frac{hp \ s \ a = \lfloor (D, fs) \rfloor \quad \neg P \vdash D \preceq^* C}{P \vdash \langle \text{try Throw } a \ \text{catch } (C \ V) \ e_2, s \rangle \rightarrow \langle \text{Throw } a, s \rangle}
\end{array}$$

Fig. 8. Exceptional expression reduction

$$\begin{array}{l}
P \vdash \langle \text{Cast } C \ (\text{throw } e), s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle V := \text{throw } e, s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \text{throw } e.F\{D\}, s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \text{throw } e.F\{D\} := e_2, s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \text{Val } v.F\{D\} := \text{throw } e, s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \text{throw } e \ll bop \gg e_2, s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \text{Val } v_1 \ll bop \gg \text{throw } e, s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \{V:T; \text{Throw } a\}, s \rangle \rightarrow \langle \text{Throw } a, s \rangle \\
P \vdash \langle \{V:T; V := \text{Val } v; \text{Throw } a\}, s \rangle \rightarrow \langle \text{Throw } a, s \rangle \\
P \vdash \langle \text{throw } e.M(es), s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \text{Val } v.M(\text{map Val } vs @ (\text{throw } e \cdot es')), s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \text{throw } e; e_2, s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \text{if } (\text{throw } e) \ e_1 \ \text{else } e_2, s \rangle \rightarrow \langle \text{throw } e, s \rangle \\
P \vdash \langle \text{throw } (\text{throw } e), s \rangle \rightarrow \langle \text{throw } e, s \rangle
\end{array}$$

Fig. 9. Exception propagation

be reduced to an address first.

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

$$\begin{aligned}
 wf\text{-prog} &:: 'm\ wf\text{-mdecl}\text{-test} \Rightarrow 'm\ prog \Rightarrow bool \\
 wf\text{-prog}\ wf\text{-md}\ P &\equiv wf\text{-syscls}\ P \wedge (\forall c \in set\ P.\ wf\text{-cdecl}\ wf\text{-md}\ P\ c) \wedge distinct\text{-fst}\ P \\
 wf\text{-syscls} &:: 'm\ prog \Rightarrow bool \\
 wf\text{-syscls}\ P &\equiv \{Object\} \cup sys\text{-xcepts} \subseteq set\ (map\ fst\ P) \\
 wf\text{-cdecl} &:: 'm\ wf\text{-mdecl}\text{-test} \Rightarrow 'm\ prog \Rightarrow 'm\ cdecl \Rightarrow bool \\
 wf\text{-cdecl}\ wf\text{-md}\ P &\equiv \\
 \lambda(C, D, fs, ms). & \\
 (\forall f \in set\ fs.\ wf\text{-fdecl}\ P\ f) \wedge distinct\text{-fst}\ fs \wedge (\forall m \in set\ ms.\ wf\text{-mdecl}\ wf\text{-md}\ P\ C\ m) \wedge & \\
 distinct\text{-fst}\ ms \wedge & \\
 (C \neq Object \longrightarrow & \\
 is\text{-class}\ P\ D \wedge \neg P \vdash D \preceq^* C \wedge & \\
 (\forall (M, Ts, T, m) \in set\ ms. & \\
 \forall D'\ Ts'\ T'\ m'. & \\
 P \vdash D\ sees\ M: Ts' \rightarrow T' = m' \text{ in } D' \longrightarrow P \vdash Ts' [\leq] Ts \wedge P \vdash T \leq T')) & \\
 wf\text{-fdecl} &:: 'm\ prog \Rightarrow fdecl \Rightarrow bool \\
 wf\text{-fdecl}\ P &\equiv \lambda(F, T).\ is\text{-type}\ P\ T \\
 wf\text{-mdecl} &:: 'm\ wf\text{-mdecl}\text{-test} \Rightarrow 'm\ wf\text{-mdecl}\text{-test} \\
 wf\text{-mdecl}\ wf\text{-md}\ P\ C &\equiv \\
 \lambda(M, Ts, T, mb). (\forall T \in set\ Ts.\ is\text{-type}\ P\ T) \wedge is\text{-type}\ P\ T \wedge wf\text{-md}\ P\ C\ (M, Ts, T, mb) &
 \end{aligned}$$

Fig. 10. Generic well-formedness

2.4 Well-formedness

We are now aiming to show that the big and small step semantics are closely related. For this (and many other proofs) we need to impose various well-formedness constraints on programs. Some of them are generic, like the constraint that the type in a field declaration must be a valid type. Others depend on the type of method bodies. To factor out the latter constraints, the well-formedness test on programs will be parameterized by a well-formedness test for methods:

$$\mathbf{types}\ 'm\ wf\text{-mdecl}\text{-test} = 'm\ prog \Rightarrow cname \Rightarrow 'm\ mdecl \Rightarrow bool$$

Tests of this type are meant to check if a certain method declaration in a certain class within a certain program is well-formed.

Declarations are lists of pairs. In order to forbid repeated declarations of the same name, we introduce the auxiliary predicate *distinct-fst* which checks that in a list of pairs all first components are distinct: $distinct\text{-fst} \equiv distinct \circ map\ fst$.

The well-formedness predicates are shown in Fig. 10. They employ the following notions from Jinja’s type system: $is\text{-type}\ P\ T$ checks if T is a valid Jinja type, $P \vdash T \leq T'$ checks if T is a subtype of T' , and $P \vdash Ts [\leq] Ts'$ if, element by element, Ts is a subtype of Ts' , all in the context of P . We will only define these notions formally in §2.6 because their definition is not relevant beforehand.

Let us now look at Fig. 10. A program is ok (= well-formed) iff it contains all system classes (*Object* and all system exceptions), all class declarations are ok, and no class is declared twice. A declaration of a class C is ok iff all field declarations and all method declarations are ok, no field or method is declared twice, and, if $C \neq Object$, then its superclass D exists, D is not a subclass of C (to rule out cycles), and method overriding is contravariant in the argument type and covariant

in the result type. That is, if C overrides a method declaration visible from D , then the new declaration must have more specific argument types and a more general result type. Note that overriding involves only the method name — there is no overloading.

2.4.1 Weak well-formedness. We will now instantiate *wf-prog* with a constraint needed for relating big and small step semantics: method bodies should not refer to global variables. This requires the notion of the **free variables** in an expression, collected by $fv :: expr \Rightarrow vname\ set$ defined in Fig. 11. A Jinja method declaration is **weakly well-formed** iff the following conditions hold: there are as many parameter types as parameter names, the parameter names are all distinct, *this* is not among the parameter names, and the free variables of the body refer only to *this* and the parameters names. Formally:

$$\begin{aligned} wwf\text{-}J\text{-}mdecl &:: J\text{-}mb\ wf\text{-}mdecl\text{-}test \\ wwf\text{-}J\text{-}mdecl\ P\ C &\equiv \\ \lambda(M, Ts, T, pns, body). & \\ |Ts| = |pns| \wedge distinct\ pns \wedge this \notin set\ pns \wedge fv\ body \subseteq \{this\} \cup set\ pns & \end{aligned}$$

The key requirement is $fv\ body \subseteq \{this\} \cup set\ pns$: it rules out reference to global variables. This is necessary to make the big and small step semantics of method call coincide. In the big step semantics, *body* is evaluated in a store containing only *this* and the parameters. In the small step semantics, *this* and the parameters are (indirectly) added to the current store, which would lead to dynamic variable binding (see §2.3.2) if *body* contained free variables outside $\{this\} \cup set\ pns$.

The condition $|Ts| = |pns|$ is necessary because we have separated parameter names from their types. Normally there is a combined parameter list of pairs (V, T) , just as in field declarations. However, since parameter names do not make sense on the machine level, but parameter types do, we have separated these two concepts in our generic type of programs.

A Jinja program is weakly well-formed iff all its method bodies are:

$$wwf\text{-}J\text{-}prog \equiv wf\text{-}prog\ wwf\text{-}J\text{-}mdecl$$

2.5 Relating Big and Small Step Semantics

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

THEOREM 2.3. If $wwf\text{-}J\text{-}prog\ P$ then
 $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$ iff $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \wedge final\ e'$.

One half of the only-if-direction of Theorem 2.3 is Lemma 2.1, the other half is

THEOREM 2.4. If $wwf\text{-}J\text{-}prog\ P$ and $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$ then
 $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$.

PROOF. The proof is by induction on \Rightarrow . Most cases follow the simple pattern that we demonstrate for **Cast**. First we lift the subexpression reduction rules from \rightarrow to \rightarrow^* , i.e. we show $P \vdash \langle e, s \rangle \rightarrow^* \langle null, s' \rangle \implies P \vdash \langle \mathbf{Cast}\ C\ e, s \rangle \rightarrow^* \langle null, s' \rangle$ which follows from rule $P \vdash \langle \mathbf{Cast}\ C\ null, s \rangle \rightarrow \langle null, s \rangle$ with the help of the lemma $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle \implies P \vdash \langle \mathbf{Cast}\ C\ e, s \rangle \rightarrow^* \langle \mathbf{Cast}\ C\ e', s' \rangle$ which is proved from rule $P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \implies P \vdash \langle \mathbf{Cast}\ C\ e, s \rangle \rightarrow \langle \mathbf{Cast}\ C\ e', s' \rangle$ by induction on \rightarrow^* . Now the proposition follows by induction hypothesis.

$$\begin{aligned}
 fv(\text{new } C) &= \{\} \\
 fv(\text{Cast } C \ e) &= fv \ e \\
 fv(\text{Val } v) &= \{\} \\
 fv(e_1 \ll bop \gg e_2) &= fv \ e_1 \cup fv \ e_2 \\
 fv(\text{Var } V) &= \{V\} \\
 fv(V := e) &= \{V\} \cup fv \ e \\
 fv(e.F\{D\}) &= fv \ e \\
 fv(e_1.F\{D\} := e_2) &= fv \ e_1 \cup fv \ e_2 \\
 fv(e.M(es)) &= fv \ e \cup fvs \ es \\
 fv\{V:T; e\} &= fv \ e - \{V\} \\
 fv(e_1; e_2) &= fv \ e_1 \cup fv \ e_2 \\
 fv(\text{if } (b) \ e_1 \ \text{else } \ e_2) &= fv \ b \cup fv \ e_1 \cup fv \ e_2 \\
 fv(\text{while } (b) \ e) &= fv \ b \cup fv \ e \\
 fv(\text{throw } e) &= fv \ e \\
 fv(\text{try } e_1 \ \text{catch } (C \ V) \ e_2) &= fv \ e_1 \cup (fv \ e_2 - \{V\}) \\
 fvs \ \square &= \{\} \\
 fvs(e \cdot es) &= fv \ e \cup fvs \ es
 \end{aligned}$$

Fig. 11. Free variables

For blocks (and similarly for try-catch), the lifting is more complicated:

$$\begin{aligned}
 [P \vdash \langle e_0, (h_0, l_0(V := \text{None})) \rangle \rightarrow^* \langle e_2, (h_2, l_2) \rangle; \text{final } e_2] \\
 \implies P \vdash \langle \{V:T; e_0\}, (h_0, l_0) \rangle \rightarrow^* \langle e_2, (h_2, l_2(V := l_0 \ V)) \rangle
 \end{aligned}$$

is proved by induction on \rightarrow^* in the premise. The induction step can be proved via

$$\begin{aligned}
 [P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow^* \langle e', (h', l') \rangle; \text{final } e'] \\
 \implies P \vdash \langle \{V:T; V := \text{Val } v; e\}, (h, l) \rangle \rightarrow^* \langle e', (h', l'(V := l \ V)) \rangle
 \end{aligned}$$

which follows easily from

$$\begin{aligned}
 P \vdash \langle e, (h, l(V \mapsto v)) \rangle \rightarrow^* \langle e', (h', l') \rangle \implies \\
 P \vdash \langle \{V:T; V := \text{Val } v; e\}, (h, l) \rangle \rightarrow^* \langle \{V:T; V := \text{Val } (\text{the } (l' \ V)); e'\}, (h', l'(V := l \ V)) \rangle
 \end{aligned}$$

which can be proved by induction on \rightarrow^* in the premise.

The most complex case is method call where we have to prove the small step simulation of the exception-free big step call rule, i.e.

$$\begin{aligned}
 [wuf\text{-}J\text{-prog } P; P \vdash \langle e, s_0 \rangle \rightarrow^* \langle \text{addr } a, s_1 \rangle; P \vdash \langle es, s_1 \rangle [\mapsto]^* \langle \text{map Val } vs, (h_2, l_2) \rangle; \\
 h_2 \ a = [(C, fs)]; P \vdash C \ \text{sees } M: Ts \rightarrow T = (pns, \text{body}) \ \text{in } D; |vs| = |pns|; \\
 l_2' = [\text{this} \mapsto \text{Addr } a, pns \ [\mapsto] \ vs]; P \vdash \langle \text{body}, (h_2, l_2') \rangle \rightarrow^* \langle ef, (h_3, l_3) \rangle; \text{final } ef] \\
 \implies P \vdash \langle e.M(es), s_0 \rangle \rightarrow^* \langle ef, (h_3, l_2) \rangle
 \end{aligned}$$

It is straightforward to derive $P \vdash \langle e.M(es), s_0 \rangle \rightarrow^* \langle (\text{addr } a).M(es), s_1 \rangle \rightarrow^* \langle (\text{addr } a).M(\text{map Val } pvs), (h_2, l_2) \rangle \rightarrow \langle \text{blks}, (h_2, l_2) \rangle$ (where *blks* abbreviates blocks (*this* · *pns*, *Class D* · *Ts*, *Addr a* · *pvs*, *body*)) from the assumptions. From $P \vdash \langle \text{body}, (h_2, l_2') \rangle \rightarrow^* \langle ef, (h_3, l_3) \rangle$ (1) it follows by the easy lemma

$$P \vdash \langle e, (h, l) \rangle \rightarrow^* \langle e', (h', l') \rangle \implies P \vdash \langle e, (h, l_0 \ ++ \ l) \rangle \rightarrow^* \langle e', (h', l_0 \ ++ \ l') \rangle$$

(provable by induction) that $P \vdash \langle \text{body}, (h_2, l_2(\text{this} \mapsto \text{Addr } a, pns \ [\mapsto] \ pvs)) \rangle \rightarrow^* \langle ef, (h_3, l_2 \ ++ \ l_3) \rangle$. Now we can transfer the bindings for *this* and *pns* from the store into blocks to obtain

$$P \vdash \langle \text{blks}, (h_2, l_2) \rangle \rightarrow^* \langle ef, (h_3, (l_2 \ ++ \ l_3)(l_2\{\text{this}\} \cup \text{set } pns)) \rangle$$

where $f(g|A)$ means $\lambda a. \text{if } a \in A \text{ then } g \text{ a else } f \text{ a}$. Finally we prove $(l_2 ++ l_3)(l_2|\{\text{this}\} \cup \text{set } pns) = l_2$ (2), which finishes the call case. The proof of (2) is easy once we know $\text{dom } l_3 \subseteq \{\text{this}\} \cup \text{set } pns$ which in turn follows from (1) using the lemma

$$\llbracket \text{wuf-J-prog } P; P \vdash \langle e, (h, l) \rangle \rightarrow^* \langle e', (h', l') \rangle \rrbracket \Longrightarrow \text{dom } l' \subseteq \text{dom } l \cup \text{fv } e$$

which in its turn is proved using the lemma

$$\llbracket \text{wuf-J-prog } P; P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \rrbracket \Longrightarrow \text{fv } e' \subseteq \text{fv } e$$

□

The other direction of Theorem 2.3

THEOREM 2.5. If $\text{wuf-J-prog } P$ 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 easily by induction on \rightarrow^* : the base case is Lemma 2.2, the induction step follows directly from

THEOREM 2.6. If $\text{wuf-J-prog } P$ 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 $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$.

It is proved by induction on \rightarrow . Most cases are straightforward, except for method call, which requires the following three lemmas:

$$\llbracket |ps| = |Ts|; |ps| = |vs|; P \vdash \langle \text{blocks } (ps, Ts, vs, e), (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle \rrbracket \\ \Longrightarrow \exists l''. P \vdash \langle e, (h, l(\text{ps } [\mapsto] \text{vs})) \rangle \Rightarrow \langle e', (h', l'') \rangle$$

$$\llbracket \text{wuf-J-prog } P; P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle; \text{fv } e \subseteq W \rrbracket \Longrightarrow P \vdash \langle e, (h, l \upharpoonright W) \rangle \Rightarrow \langle e', (h', l' \upharpoonright W) \rangle$$

$$\llbracket P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle; \text{fv } e = \{\} \rrbracket \Longrightarrow l' = l$$

The notation $m \upharpoonright_A$ means restriction of m to A , i.e. $\lambda x. \text{if } x \in A \text{ then } m \ x \text{ else None}$. The proofs of these lemmas and the proof of the method call case from them are reasonably straightforward.

Note that the fact that Theorem 2.6 holds is not just a nice coincidence: this theorem is trivially implied whenever \Rightarrow and \rightarrow^* coincide.

2.6 Type System

Having concluded the dynamic semantics, we now turn to context conditions, starting with the type system. Jinja types are either primitive (*Boolean* and *Integer*), class types *Class* C , *NT* (the type of *Null*), or *Void* (the type of *Unit*). The corresponding HOL type is called *ty*. A **reference type** is either *Class* C or *NT* — predicate $\text{is-refT} :: \text{ty} \Rightarrow \text{bool}$ tests for reference types.

Types should only refer to classes that exist in the current program:

$$\text{is-type } P \ T \equiv \text{case } T \text{ of Class } C \Rightarrow \text{is-class } P \ C \mid - \Rightarrow \text{True}$$

Function $\text{typeof} :: \text{heap} \Rightarrow \text{val} \Rightarrow \text{ty option}$ computes the type of a value. The heap argument is necessary because values may contain addresses. The result type is *ty option* rather than *ty* because unallocated addresses do not have a type.

$$\text{typeof}_h \ \text{Unit} = \lfloor \text{Void} \rfloor \\ \text{typeof}_h \ \text{Null} = \lfloor \text{NT} \rfloor$$

$$\begin{aligned} \text{typeof}_h(\text{Bool } b) &= [\text{Boolean}] \\ \text{typeof}_h(\text{Intg } i) &= [\text{Integer}] \\ \text{typeof}_h(\text{Addr } a) &= (\text{case } h \text{ a of } \text{None} \Rightarrow \text{None} \mid [(C, fs)] \Rightarrow [\text{Class } C]) \end{aligned}$$

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

$$\text{typeof } v \equiv \text{typeof}_{\text{empty}} v$$

The subclass relation $P \vdash C \preceq^* C'$ induces a **subtype** relation $P \vdash T \leq T'$ (often called *widening*) in the obvious manner:

$$P \vdash T \leq T \quad P \vdash NT \leq \text{Class } C \quad \frac{P \vdash C \preceq^* D}{P \vdash \text{Class } C \leq \text{Class } D}$$

The pointwise extension of \leq to lists of types is written $[\leq]$.

The core of the type system is the judgement $P, E \vdash e :: T$, where E is an **environment**, i.e. a map from variables to their types. The complete set of typing rules is shown in Fig. 12. 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.

Now we examine the remaining rules for $P, E \vdash e :: T$. We only allow up and down casts: other casts are pointless because they are bound to fail at runtime. The rules for $e_1 \ll=\gg e_2$ and **if** (e) e_1 **else** e_2 follow Java (with its conditional operator $?$: rather than its if-else) in requiring that the type of e_1 is a subtype of that of e_2 or conversely. Loops are of type *Void* because they evaluate to *unit*. Exceptions (**throw**) are of type *Void*, as in Java. They could also be polymorphic, but that would complicate the type system. The rule for **try** e_1 **catch** (C V) e_2 follows Java (where e_1 and e_2 must be statements) in requiring that e_1 and e_2 have the same type.

The extension of $::$ to lists is denoted by $[\:]$.

Although the rules for $P, E \vdash e :: T$ can be viewed as computing the annotations $\{D\}$ (via the constraint on D), an explicit computation $P, E \vdash e \rightsquigarrow e'$ may be clearer: the input e is an unannotated expression, the output e' its annotated version. Here are two representative rules, one that just copies, and one that adds an annotation:

$$\frac{P, E \vdash e \rightsquigarrow e'}{P, E \vdash \text{Cast } C \ e \rightsquigarrow \text{Cast } C \ e'}$$

$$\frac{P, E \vdash e \rightsquigarrow e' \quad P, E \vdash e' :: \text{Class } C \quad P \vdash C \text{ sees } F:T \text{ in } D}{P, E \vdash e.F \rightsquigarrow e'.F\{D\}}$$

While we are at it, we also determine if some **Var** V really refers to a variable or to a field. In the latter case it is prefixed by *this* and annotated:

$$\frac{E \ V = [T]}{P, E \vdash \text{Var } V \rightsquigarrow \text{Var } V} \quad \frac{E \ V = \text{None} \quad E \ \text{this} = [\text{Class } C] \quad P \vdash C \text{ sees } V:T \text{ in } D}{P, E \vdash \text{Var } V \rightsquigarrow \text{Var } \text{this}.V\{D\}}$$

Thus \rightsquigarrow should be viewed as a translation that is part of type checking. As we do not refer to it again, we need not show the remaining (obvious) rules.

$$\begin{array}{c}
\frac{\text{is-class } P \ C}{P, E \vdash \mathbf{new} \ C :: \text{Class } C} \quad \frac{\text{typeof } v = [T]}{P, E \vdash \mathbf{val} \ v :: T} \quad \frac{E \ V = [T]}{P, E \vdash \mathbf{var} \ V :: T} \\
\frac{P, E \vdash e :: \text{Class } D \quad \text{is-class } P \ C \quad P \vdash C \preceq^* D \vee P \vdash D \preceq^* C}{P, E \vdash \mathbf{cast} \ C \ e :: \text{Class } C} \\
\frac{P, E \vdash e_1 :: T_1 \quad P, E \vdash e_2 :: T_2 \quad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1}{P, E \vdash e_1 \ll = \gg e_2 :: \text{Boolean}} \\
\frac{P, E \vdash e_1 :: \text{Integer} \quad P, E \vdash e_2 :: \text{Integer}}{P, E \vdash e_1 \ll + \gg e_2 :: \text{Integer}} \\
\frac{E \ V = [T] \quad P, E \vdash e :: T' \quad P \vdash T' \leq T \quad V \neq \mathbf{this}}{P, E \vdash V := e :: \text{Void}} \\
\frac{P, E \vdash e :: \text{Class } C \quad P \vdash C \ \text{sees } F:T \ \text{in } D}{P, E \vdash e.F\{D\} :: T} \\
\frac{P, E \vdash e_1 :: \text{Class } C \quad P \vdash C \ \text{sees } F:T \ \text{in } D \quad P, E \vdash e_2 :: T' \quad P \vdash T' \leq T}{P, E \vdash e_1.F\{D\} := e_2 :: \text{Void}} \\
\frac{P, E \vdash e :: \text{Class } C \quad P \vdash C \ \text{sees } M: T_s \rightarrow T = (\text{pns}, \text{body}) \ \text{in } D \quad P, E \vdash es \ [::] \ T_s' \quad P \vdash T_s' \ [\leq] \ T_s}{P, E \vdash e.M(es) :: T} \\
\frac{\text{is-type } P \ T \quad P, E(V \mapsto T) \vdash e :: T'}{P, E \vdash \{V:T; e\} :: T'} \quad \frac{P, E \vdash e_1 :: T_1 \quad P, E \vdash e_2 :: T_2}{P, E \vdash e_1; e_2 :: T_2} \\
\frac{P, E \vdash e :: \text{Boolean}}{P, E \vdash e_1 :: T_1 \quad P, E \vdash e_2 :: T_2 \quad P \vdash T_1 \leq T_2 \vee P \vdash T_2 \leq T_1} \\
\frac{P \vdash T_1 \leq T_2 \longrightarrow T = T_2 \quad P \vdash T_2 \leq T_1 \longrightarrow T = T_1}{P, E \vdash \mathbf{if} \ (e) \ e_1 \ \mathbf{else} \ e_2 :: T} \\
\frac{P, E \vdash e :: \text{Boolean} \quad P, E \vdash c :: T}{P, E \vdash \mathbf{while} \ (e) \ c :: \text{Void}} \quad \frac{P, E \vdash e :: \text{Class } C}{P, E \vdash \mathbf{throw} \ e :: \text{Void}} \\
\frac{P, E \vdash e_1 :: T \quad P, E(V \mapsto \text{Class } C) \vdash e_2 :: T \quad \text{is-class } P \ C}{P, E \vdash \mathbf{try} \ e_1 \ \mathbf{catch} \ (C \ V) \ e_2 :: T} \\
P, E \vdash [] \ [::] \ [] \quad \frac{P, E \vdash e :: T \quad P, E \vdash es \ [::] \ T_s}{P, E \vdash e \cdot es \ [::] \ T \cdot T_s}
\end{array}$$

Fig. 12. Typing rules

2.7 Definite Assignment

One of Java’s notable features is the check that all variables must be assigned to before use, called “definite assignment”. Jinja’s rules for definite assignment are much simpler than Java’s, thus missing certain cases, but still demonstrating the feature in its full generality. We employ two recursive functions

$$\mathcal{D} :: \text{expr} \Rightarrow \text{vname set option} \Rightarrow \text{bool} \quad \mathcal{A} :: \text{expr} \Rightarrow \text{vname set option}$$

For a moment ignore the *option* type. Then $\mathcal{D} \ e \ A$ is meant to check if evaluation of e starting from any state where all variables in A are initialized only accesses initialized variables. The auxiliary function $\mathcal{A} \ e$ computes the set of variables that have been assigned to after any normal evaluation of e .

$$\begin{aligned}
 \mathcal{D}(\text{new } C) A &= \text{True} \\
 \mathcal{D}(\text{Cast } C e) A &= \mathcal{D} e A \\
 \mathcal{D}(\text{Val } v) A &= \text{True} \\
 \mathcal{D}(e_1 \ll \text{bop} \gg e_2) A &= (\mathcal{D} e_1 A \wedge \mathcal{D} e_2 (A \sqcup \mathcal{A} e_1)) \\
 \mathcal{D}(\text{Var } V) A &= (V \in\in A) \\
 \mathcal{D}(V := e) A &= \mathcal{D} e A \\
 \mathcal{D}(e.F\{D\}) A &= \mathcal{D} e A \\
 \mathcal{D}(e_1.F\{D\} := e_2) A &= (\mathcal{D} e_1 A \wedge \mathcal{D} e_2 (A \sqcup \mathcal{A} e_1)) \\
 \mathcal{D}(e.M(es)) A &= (\mathcal{D} e A \wedge \mathcal{D} s es (A \sqcup \mathcal{A} e)) \\
 \mathcal{D}\{V:T; e\} A &= \mathcal{D} e (A \ominus V) \\
 \mathcal{D}(e_1; e_2) A &= (\mathcal{D} e_1 A \wedge \mathcal{D} e_2 (A \sqcup \mathcal{A} e_1)) \\
 \mathcal{D}(\text{if } (e) e_1 \text{ else } e_2) A &= (\mathcal{D} e A \wedge \mathcal{D} e_1 (A \sqcup \mathcal{A} e) \wedge \mathcal{D} e_2 (A \sqcup \mathcal{A} e)) \\
 \mathcal{D}(\text{while } (e) e) A &= (\mathcal{D} e A \wedge \mathcal{D} c (A \sqcup \mathcal{A} e)) \\
 \mathcal{D}(\text{throw } e) A &= \mathcal{D} e A \\
 \mathcal{D}(\text{try } e_1 \text{ catch } (C V) e_2) A &= (\mathcal{D} e_1 A \wedge \mathcal{D} e_2 (A \sqcup \{\{V\}\})) \\
 \mathcal{D} s \square A &= \text{True} \\
 \mathcal{D} s (e \cdot es) A &= (\mathcal{D} e A \wedge \mathcal{D} s es (A \sqcup \mathcal{A} e))
 \end{aligned}$$

 Fig. 13. Definition of \mathcal{D}

$$\begin{aligned}
 \mathcal{A}(\text{new } C) &= \{\{\}\} \\
 \mathcal{A}(\text{Cast } C e) &= \mathcal{A} e \\
 \mathcal{A}(\text{Val } v) &= \{\{\}\} \\
 \mathcal{A}(e_1 \ll \text{bop} \gg e_2) &= \mathcal{A} e_1 \sqcup \mathcal{A} e_2 \\
 \mathcal{A}(\text{Var } V) &= \{\{\}\} \\
 \mathcal{A}(V := e) &= \{\{V\}\} \sqcup \mathcal{A} e \\
 \mathcal{A}(e.F\{D\}) &= \mathcal{A} e \\
 \mathcal{A}(e_1.F\{D\} := e_2) &= \mathcal{A} e_1 \sqcup \mathcal{A} e_2 \\
 \mathcal{A}(e.M(es)) &= \mathcal{A} e \sqcup \mathcal{A} s es \\
 \mathcal{A}\{V:T; e\} &= \mathcal{A} e \ominus V \\
 \mathcal{A}(e_1; e_2) &= \mathcal{A} e_1 \sqcup \mathcal{A} e_2 \\
 \mathcal{A}(\text{if } (e) e_1 \text{ else } e_2) &= \mathcal{A} e \sqcup \mathcal{A} e_1 \sqcap \mathcal{A} e_2 \\
 \mathcal{A}(\text{while } (b) e) &= \mathcal{A} b \\
 \mathcal{A}(\text{throw } e) &= \text{None} \\
 \mathcal{A}(\text{try } e_1 \text{ catch } (C V) e_2) &= \mathcal{A} e_1 \sqcap (\mathcal{A} e_2 \ominus V) \\
 \mathcal{A} s \square &= \{\{\}\} \\
 \mathcal{A} s (e \cdot es) &= \mathcal{A} e \sqcup \mathcal{A} s es
 \end{aligned}$$

 Fig. 14. Definition of \mathcal{A}

The need for *option* arises because of **throw**: what should $\mathcal{A}(\text{throw } e)$ return? We use *None* to indicate that the expression will always fail. However, you should already think of *set option* as a completion of *set* with a top element *None* representing the universal set of all variable names (called *UNIV*). We will discuss this issue again below, but first we look at the definition of \mathcal{D} (Fig. 13) and \mathcal{A} (Fig. 14). They use the auxiliary operations \sqcup , \sqcap , \ominus , \sqsubseteq and $\in\in$ (Fig. 15) which extend \cup , \cap , $-$ and \subseteq from *set* to *set option*, treating *None* (almost) as the universal set.

The rules for \mathcal{D} and \mathcal{A} are essentially straightforward and we just discuss a few of the ones for \mathcal{A} : evaluating **if** $(e) e_1$ **else** e_2 guarantees all assignments that e guarantees, and those that both e_1 and e_2 guarantee; evaluating **while** $(b) e$ guarantees only the assignments in b because e may never be evaluated; evaluating $\{V:T; e\}$ cannot assign to V , because its V is local. The test $\mathcal{D} e A$ is even more

$$\begin{aligned}
A \sqcup B &\equiv \text{case } A \text{ of } None \Rightarrow None \mid [A] \Rightarrow \text{case } B \text{ of } None \Rightarrow None \mid [B] \Rightarrow [A \cup B] \\
A \sqcap B &\equiv \text{case } A \text{ of } None \Rightarrow B \mid [A] \Rightarrow \text{case } B \text{ of } None \Rightarrow [A] \mid [B] \Rightarrow [A \cap B] \\
A \ominus a &\equiv \text{case } A \text{ of } None \Rightarrow None \mid [A] \Rightarrow [A - \{a\}] \\
A \sqsubseteq B &\equiv \text{case } B \text{ of } None \Rightarrow True \mid [B] \Rightarrow \text{case } A \text{ of } None \Rightarrow False \mid [A] \Rightarrow A \sqsubseteq B \\
a \in\in A &\equiv \text{case } A \text{ of } None \Rightarrow True \mid [A] \Rightarrow a \in A
\end{aligned}$$
Fig. 15. Operations on *set option*

uniform. It descends into all subexpressions of e in the order they are evaluated, extends A according to \mathcal{A} , and every time it comes across some $\text{Var } V$ it checks if $V \in\in A$, i.e. if V has definitely been assigned to beforehand.

Monotonicity of \mathcal{D} is proved by induction on e :

LEMMA 2.7. If $\mathcal{D} e A$ and $A \sqsubseteq A'$ then $\mathcal{D} e A'$.

Let us now come back to the question of *None* vs. *UNIV*. If we worked with *set* as opposed to *set option*, we would need to define $\mathcal{A}(\text{throw } _) = \text{UNIV}$ to ensure that $\mathcal{A}(\text{if } (e) \text{ throw } _ \text{ else } e_2) = \mathcal{A} e \cup \mathcal{A} e_2$, i.e. that $\mathcal{A}(\text{throw } _)$ does not reduce the overall result, which should only reflect normal evaluations. Returning *UNIV* in the case of guaranteed **throw** is what Schirmer [2003] does. For a start, this loses direct executability, as *UNIV* is not a finite set. What is worse, in Jinja it would lead to undesirable imprecision of the analysis. Let e be the expression $\text{if } (\text{Var } B) \{V:T; \text{throw } _ \} \text{ else } V := \text{true}$. We would obtain $\mathcal{A} \{V:T; \text{throw } _ \} = \text{UNIV} - \{V\}$, $\mathcal{A} e = \{\}$, and hence $\neg \mathcal{D}(e; V := \text{Var } V) \{B\}$, contrary to what one would expect. This is where *None* comes in: we would like $\text{UNIV} - \{V\} = \text{UNIV}$, but since that does not hold, we introduce *None* as the new top element of the lattice and define \ominus such that $\text{None} \ominus V = \text{None}$. Schirmer [2003] gets away with *UNIV* because he does not allow e above: it contains both a local and a global V , something that Java forbids. In Jinja, however, we did not want to ban nested declarations of the same variable, which cripples the classic block structure.

2.8 Well-formed Jinja Programs

Well-formedness of Jinja method declarations

$$\begin{aligned}
wf\text{-}J\text{-mdecl } P C (M, Ts, T, pns, \text{body}) &\equiv \\
|Ts| = |pns| \wedge \text{distinct } pns \wedge \text{this} \notin \text{set } pns \wedge \\
(\exists T'. P, [\text{this} \mapsto \text{Class } C, pns \mapsto Ts] \vdash \text{body} :: T' \wedge P \vdash T' \leq T) \wedge \\
\mathcal{D} \text{body } [\{\text{this}\} \cup \text{set } pns]
\end{aligned}$$

extends weak well-formedness (§2.4.1) by requiring that method bodies a) are well-typed with a subtype of the declared return type, and b) pass the definite assignment test assuming only *this* and the parameters are initialized. A Jinja program is well-formed iff all its method bodies are:

$$wf\text{-}J\text{-prog} \equiv wf\text{-}prog \text{ } wf\text{-}J\text{-mdecl}$$

2.9 Type Safety

In §2.9 we prove type safety in the traditional syntactic way [Wright and Felleisen 1994]: 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). These inductive proofs need a number of new notions (invariants) to go through.

2.9.1 Conformance. It expresses that semantic objects conform to their syntactic description. Let v_m be a map to values and T_m one to types:

Conformance of values to types.

$$P, h \vdash v : \leq T \equiv \exists T'. \text{typeof}_h v = [T'] \wedge P \vdash T' \leq T$$

Conformance of fields to types.

$$P, h \vdash v_m (: \leq) T_m \equiv \forall FD T. T_m FD = [T] \longrightarrow (\exists v. v_m FD = [v] \wedge P, h \vdash v : \leq T)$$

Weak conformance of local variables to types.

$$P, h \vdash v_m (: \leq)_w T_m \equiv \forall V v. v_m V = [v] \longrightarrow (\exists T. T_m V = [T] \wedge P, h \vdash v : \leq T)$$

Conformance of objects.

$$P, h \vdash \text{obj } \surd \equiv \text{let } (C, v_m) = \text{obj in } \exists \text{FDTs}. P \vdash C \text{ has-fields FDTs} \wedge P, h \vdash v_m (: \leq) \text{ map-of FDTs}$$

Conformance of heaps.

$$P \vdash h \surd \equiv (\forall a \text{ obj}. h a = [\text{obj}] \longrightarrow P, h \vdash \text{obj } \surd) \wedge \text{preallocated } h \\ \text{preallocated } h \equiv \forall C \in \text{sys-xcpts}. \exists \text{fs}. h (\text{addr-of-sys-xcpt } C) = [(C, \text{fs})]$$

Conformance of states.

$$P, E \vdash s \surd \equiv \text{let } (h, l) = s \text{ in } P \vdash h \surd \wedge P, h \vdash l (: \leq)_w E$$

Note that $(: \leq)$ says that all declared fields must have values of the right type, whereas $(: \leq)_w$ says that only initialized variables must have values of the right type. This reflects the difference in initialization of fields and local variables.

2.9.2 Runtime Type System. The proof of subject reduction requires a modified type system. The purpose of $\vdash ::$ (Fig. 12) is to rule out not just unsafe 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** [Drossopoulou and Eisenbach 1999] 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 rule for field access: $P, E \vdash e.F\{D\} :: T$ requires $P \vdash C \text{ sees } F:T \text{ in } D$ if e is of class C . If e reduces to an object belonging to a subclass of C , this condition may no longer be met. Thus we relax it to $P \vdash C \text{ has } F:T \text{ in } D$ which is preserved by reduction and is still strong enough to imply type safety. It is interesting to note that this change was missed by Flatt et al. [1999], which invalidates their Lemma 6 and thus subject reduction. Please keep in mind that the runtime type system is a purely technical device needed in the proof of type safety.

In Fig. 16 we show only those rules for $\vdash :$ (and $\vdash [:]$) that differ from their $::$ -counterpart beyond the addition of h . The most frequent change is the following.

$$\begin{array}{c}
\frac{\text{typeof}_h v = \lfloor T \rfloor}{P, E, h \vdash \mathbf{Val} v : T} \quad \frac{P, E, h \vdash e : T \quad \text{is-refT } T \quad \text{is-class } P C}{P, E, h \vdash \mathbf{Cast } C e : \text{Class } C} \\
\frac{P, E, h \vdash e_1 : T_1 \quad P, E, h \vdash e_2 : T_2}{P, E, h \vdash e_1 \ll \Rightarrow e_2 : \text{Boolean}} \quad \frac{E V = \lfloor T \rfloor \quad P, E, h \vdash e : T' \quad P \vdash T' \leq T}{P, E, h \vdash V := e : \text{Void}} \\
\frac{P, E, h \vdash e : \text{Class } C \quad P \vdash C \text{ has } F:T \text{ in } D}{P, E, h \vdash e.F\{D\} : T} \quad \frac{P, E, h \vdash e : NT}{P, E, h \vdash e.F\{D\} : T} \\
\frac{P, E, h \vdash e_1 : \text{Class } C \quad P \vdash C \text{ has } F:T \text{ in } D \quad P, E, h \vdash e_2 : T_2 \quad P \vdash T_2 \leq T}{P, E, h \vdash e_1.F\{D\} := e_2 : \text{Void}} \\
\frac{P, E, h \vdash e_1 : NT \quad P, E, h \vdash e_2 : T_2}{P, E, h \vdash e_1.F\{D\} := e_2 : \text{Void}} \quad \frac{P, E, h \vdash e : NT \quad P, E, h \vdash es \lfloor \rfloor Ts}{P, E, h \vdash e.M(es) : T} \\
\frac{P, E(V \mapsto T), h \vdash e : T'}{P, E, h \vdash \{V:T; e\} : T'} \quad \frac{P, E, h \vdash e : T_r \quad \text{is-refT } T_r}{P, E, h \vdash \mathbf{throw } e : T} \\
\frac{P, E, h \vdash e_1 : T_1 \quad P, E(V \mapsto \text{Class } C), h \vdash e_2 : T_2 \quad P \vdash T_1 \leq T_2}{P, E, h \vdash \mathbf{try } e_1 \mathbf{catch } (C V) e_2 : T_2}
\end{array}$$

Fig. 16. Core of runtime typing rules

Expressions that are required to be of class type by $::$ may reduce to *null*. In order to preserve well-typedness we have to add rules for the case $e :: NT$ in $e.F\{D\}$, $e.F\{D\} := e_2$ and $e.M(es)$. Note that we lose uniqueness of typing: $null.F\{D\}$ unavoidably does not even have a unique least type anymore. A similar situation arises with $\mathbf{throw } e$ and $\mathbf{Cast } C e$ where we avoid an additional rule by requiring e to be of reference type (which includes NT). For $\mathbf{try } e_1 \mathbf{catch } (C V) e_2$ we no longer require e_1 and e_2 to have the same type because reduction of e_1 may also have reduced its type. Then there is the change from *sees* to *has* for field access and update. And finally we drop two preconditions in the rules for $V := e$ and $\{V:T; e\}$ just to show that they are orthogonal to type safety.

2.9.3 The type safety proof. Under suitable conditions we can now show progress:

LEMMA 2.8. (Progress) If *wwf-J-prog* P and $P \vdash h \checkmark$ and $P, E, h \vdash e : T$ and $\mathcal{D} e \lfloor \text{dom } l \rfloor$ and $\neg \text{final } e$ then $\exists e' s'. P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', s' \rangle$.

The proof is by induction on $P, E, h \vdash e : T$. Because of the special treatment of $\{V:T; V := \mathbf{Val} v; _ \}$ we need a slightly modified induction scheme with a separate rule for this case. Alternatively one can do an induction on the size of e .

Let us examine the necessity for the individual premises. Weak well-formedness of P is needed to ensure that each method declaration has as many parameter types as parameter names, a precondition of the method call rule. In addition well-formedness is necessary for the following subtle reason: even if P defines a class C , relations *has-fields* (needed for the reduction of \mathbf{new}) and *sees* (needed for the 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 ($P \vdash h \checkmark$) is needed because otherwise an object may not have all the fields of its class and field access may get stuck. Definite assignment is required to ensure that variable access does not get stuck.

Eventually we show that a sequence of reductions preserves well-typedness by showing that each reduction preserves well-typedness. However, preservation of well-typedness 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. 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 2.9. If $P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle$ and $P, E, h \vdash e : T$ and $P \vdash h \checkmark$ then $P \vdash h' \checkmark$.

THEOREM 2.10. If $P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle$ and $P, E, h \vdash e : T$ and $P, h \vdash l (\leq)_w E$ then $P, h' \vdash l' (\leq)_w E$.

THEOREM 2.11. If *wf-J-prog* P and $P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle$ and $\mathcal{D} e \lfloor \text{dom } l \rfloor$ then $\mathcal{D} e' \lfloor \text{dom } l' \rfloor$.

Because preservation of definite assignment has not been treated in the literature before, we look at a typical case in detail: sequential composition. We assume $\mathcal{D} (e; e_2) \lfloor \text{dom } l \rfloor$, i.e. $\mathcal{D} e \lfloor \text{dom } l \rfloor \wedge \mathcal{D} e_2 (\lfloor \text{dom } l \rfloor \sqcup \mathcal{A} e)$ and have to show $\mathcal{D} (e'; e_2) \lfloor \text{dom } l' \rfloor$, i.e. $\mathcal{D} e' \lfloor \text{dom } l' \rfloor \wedge \mathcal{D} e_2 (\lfloor \text{dom } l' \rfloor \sqcup \mathcal{A} e')$. From $\mathcal{D} e \lfloor \text{dom } l \rfloor$ it follows by induction hypothesis that $\mathcal{D} e' \lfloor \text{dom } l' \rfloor$, and from $\mathcal{D} e_2 (\lfloor \text{dom } l \rfloor \sqcup \mathcal{A} e)$ it follows with lemma

$$P \vdash \langle e, (h, l) \rangle \rightarrow \langle e', (h', l') \rangle \implies \lfloor \text{dom } l \rfloor \sqcup \mathcal{A} e \sqsubseteq \lfloor \text{dom } l' \rfloor \sqcup \mathcal{A} e'$$

(which is proved by induction over \rightarrow) together with monotonicity of \mathcal{D} that $\mathcal{D} e_2 (\lfloor \text{dom } l' \rfloor \sqcup \mathcal{A} e')$, thus concluding the case.

The main preservation theorem is single step subject reduction:

THEOREM 2.12. If *wf-J-prog* P and $P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle$ and $P, E \vdash s \checkmark$ and $P, E, hp \ s \vdash e : T$ then $\exists T'. P, E, hp \ s' \vdash e' : T' \wedge P \vdash T' \leq T$.

The proof is again by induction on \rightarrow .

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

$$P, E, s \vdash e : T \checkmark \equiv P, E \vdash s \checkmark \wedge P, E, hp \ s \vdash e : T$$

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

$$\begin{aligned} & \llbracket \text{wf-J-prog } P; P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle; P, E, s \vdash e : T \checkmark \rrbracket \\ & \implies \exists T'. P, E, s' \vdash e' : T' \checkmark \wedge P \vdash T' \leq T \end{aligned}$$

Induction yields the final form of subject reduction:

THEOREM 2.13. If *wf-J-prog* P and $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$ and $P, E, s \vdash e : T \checkmark$ then $\exists T'. P, E, s' \vdash e' : T' \checkmark \wedge P \vdash T' \leq T$.

Combining this theorem, the extension of Theorem 2.11 to \rightarrow^* , and progress, and replacing the runtime type system by the original one, yields

COROLLARY 2.14. (Type Safety)

If *wf-J-prog* P and $P, E \vdash s \checkmark$ and $P, E \vdash e :: T$ and $\mathcal{D} e \lfloor \text{dom } (lcl \ s) \rfloor$ and $P \vdash \langle e, s \rangle \rightarrow^* \langle e', s' \rangle$ and $\nexists e'' \ s''. P \vdash \langle e', s' \rangle \rightarrow \langle e'', s'' \rangle$ then $(\exists v. e' = \mathbf{Val} \ v \wedge P, hp \ s' \vdash v : \leq T) \vee (\exists a. e' = \mathbf{Throw} \ a \wedge a \in \text{dom} (hp \ s'))$.

If the program and the initial state are ok, and the expression is well-typed (w.r.t. $\vdash ::$) and has the definite assignment property, then reduction to normal form yields either a value of a subtype of the initial expression or throws an existing object.

Note that we intentionally leave out the details of many of these proofs because type safety proofs abound in the literature.

2.10 Related Work

Most closely related are the work by Nipkow and Oheimb [1998; 1999] which provided the starting point for our big step semantics. Also closely related is the small step semantics by Drossopoulou and Eisenbach [1999]. The main difference is that they distinguish three very similar languages and transform from one into the next, whereas we have intentionally started with the “enriched language” (hence the class annotations in field references) and have taken care to identify it with the “runtime language”. There are also a number of smaller differences (e.g. we have omitted interfaces and arrays for space reasons, although they were present in our previous work [Nipkow and Oheimb 1998]), in particular the way in which method calls are unfolded: we use local variables to hold the parameter values whereas they choose new names and add those to the store. A name is new if it is not yet in the domain of the store. This scheme would need to be refined in a language like Jinja where one can have uninitialised local variables. Also closely related is the small step semantics by Flatt et al. [1999] who define a much smaller subset of Java without mutable variables. Nobody seems to have connected these two styles of Java semantics before, except Nipkow [2005], who obtained an even closer correspondence, but at the cost of allowing dynamic binding in the big step semantics, too. Syme [1999] formalised the work by Drossopoulou and Eisenbach [1999] in his theorem prover DECLARE. The main difference is that his store is a list of maps, one for each method invocation. DECLARE, just like Isar [Wenzel 2002], aims at readable proofs. Ancona et al. [2001] analyse a feature of Java we have ignored, namely the possibility to declare which exceptions a method may raise. This leads to very subtle interactions between type system and exceptions. Schirmer [2004] has analysed the interaction of packages and access modifiers. On the other end of the spectrum we have Featherweight Java [Igarashi et al. 2001], a minimal subset of Java which was used to study type soundness proofs. On the borderline to program verification we have the denotational semantics of Java by Huisman [2001]. Further aspects formalised in the literature but beyond this paper include multi threading, dynamic class loading, inner classes, generic classes and mixins.

3. JINJA VIRTUAL MACHINE

This section presents the machine model of the Jinja Virtual Machine (§3.1) and its operational semantics, first without (§3.2) and then with (§3.3) runtime type checks.

3.1 Machine Model

The model of the Jinja VM comprises the state space and the definition of method bodies. The state space of the Jinja VM is modelled closely after the Java VM.

The state consists of a heap, a stack of call frames, and a flag whether an exception was raised (and, if yes, a reference to the exception object).

types $jvm\text{-state} = addr\ option \times heap \times frame\ list$

The heap is the same as in the source language. The exception flag corresponds to the expression `throw` in the source language. The frame list is new.

Each method execution gets its own call frame, containing its own operand stack (a list of values), its own set of registers² for local variables (also a list of values), and its own program counter. We also store the class and name of the method and arrive at:

types $frame = opstack \times registers \times cname \times mname \times pc$
 $opstack = val\ list$
 $registers = val\ list$

It will turn out that the list of local variables is of fixed length, i.e. does not change with program execution. In fact, it is only modified and accessed by updating and indexing it at some position. Hence it can be implemented as an array. The size of the operand stack may change during execution, but here the maximum size is known statically. This enables efficient implementation for the stack as well. Although the registers do not exclusively store the local variables of the method, but also its parameters and *this*-pointer, we use the terms *registers* and *local variables* interchangeably when the distinction is not important or clear from context.

The instruction set of the Jinja VM is listed in Fig. 17. The instructions are a bit more abstract than comparable Java VM instructions, but no further conceptual simplifications have been made. In Java, there is for instance a separate `Load` instruction for most of the basic machine types, in Jinja there is only one polymorphic instruction. The more high-level instructions that may seem to be a substantial simplification over a real machine on the other hand (like `Getfield`, `Putfield` and `Invoke`), have a direct correspondence to instructions in the Java VM.

Method bodies are lists of instructions together with an exception table and two numbers mxs and mxl_0 . These are the maximum operand stack size and the number of local variables (not counting the *this* pointer and the parameters of the method, which are stored in the first 0 to n registers). So the type parameter ' m ' for method bodies gets instantiated with $nat \times nat \times instr\ list \times ex\ table$:

types $jvm\text{-method} = nat \times nat \times instr\ list \times ex\ table$
 $jvm\text{-prog} = jvm\text{-method}\ prog$

The exception table is a list of tuples (f, t, C, h, d) :

types $ex\ table = (nat \times nat \times cname \times nat \times nat)\ list$

The asymmetric interval $[f, t)$ denotes those instructions in the method body that correspond to the *try* block on the source level. The handler pc h points to the first instruction of the corresponding *catch* block. The code starting at h is the

²We deliberately deviate here from the nomenclature of the JVM specification that calls these *local variables*. They hold the *this*-pointer, the parameters, and what would be the local variables in the source language. It will be important when talking about the compiler below to distinguish between these.

datatype <i>instr</i> =		
Load <i>nat</i>		load from register
Store <i>nat</i>		store into register
Push <i>val</i>		push a constant
New <i>cname</i>		create object on heap
Getfield <i>vname cname</i>		fetch field from object
Putfield <i>vname cname</i>		set field in object
Checkcast <i>cname</i>		check if object is of class <i>cname</i>
Invoke <i>mname nat</i>		invoke instance method with <i>nat</i> parameters
Return		return from method
Pop		remove top element
IAdd		integer addition
Goto <i>int</i>		goto relative address
CmpEq		equality comparison
IfFalse <i>int</i>		branch if top of stack false
Throw		throw exception

Fig. 17. The Jinja bytecode instruction set.

exception handler, and d is the size of the stack the exception handler expects. An exception handler **protects** a program position pc iff $pc \in [f, t)$. An exception table entry **matches** an exception E if the handler protects the current pc and if the class of E is a subclass of C .

3.2 Operational Semantics

This section defines the state transition relation of the Jinja VM.

For easy direct executability, our main definition of the operational semantics of the Jinja VM is written down in a functional rather than in a relational style. The function $exec :: jvm\text{-}prog \Rightarrow jvm\text{-}state \Rightarrow jvm\text{-}state\ option$ describes one-step execution:

$$\begin{aligned}
 exec\ P\ (xp, h, []) &= None \\
 exec\ P\ ([a], h, frs) &= None \\
 exec\ P\ (None, h, (stk, loc, C, M, pc) \cdot frs) &= \\
 (\text{let } i = (instrs\text{-}of\ P\ C\ M)_{[pc]}; (xp', h', frs') = exec\text{-}instr\ i\ P\ h\ stk\ loc\ C\ M\ pc\ frs \\
 \text{in } [case\ xp' \text{ of } None \Rightarrow (None, h', frs') \\
 \quad | [a] \Rightarrow find\text{-}handler\ P\ a\ h\ ((stk, loc, C, M, pc) \cdot frs)])
 \end{aligned}$$

It says that execution halts if the call frame stack is empty or an unhandled exception has occurred. In all other cases, execution is defined: $exec$ decomposes the top call frame, retrieves the instruction list of the current method via $instrs\text{-}of$, delegates actual execution for single instructions to $exec\text{-}instr$, and finally sets the pc to the appropriate exception handler (with $find\text{-}handler$) if an exception occurred.

The function $instrs\text{-}of$ selects the instruction sequence of method M in class C of program P . As the operational semantics of the Jinja VM at this level is phrased in a functional rather than a relational style, we turn the field and method accessor relations of §2.1.4 into functions. To look up methods, we use $method$, to look up fields, we use $field$. They satisfy:

$$P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \implies method\ P\ C\ M = (D, Ts, T, m)$$

$$P \vdash C \text{ sees } F: T \text{ in } D \implies field\ P\ C\ F = (D, T)$$

Exception handling in *find-handler* (definition omitted) is similar to the Java VM: it looks up the exception table in the current method, and sets the program counter to the first handler that protects *pc* and that matches the exception class. If there is no such handler, the topmost call frame is popped, and the search continues recursively in the invoking frame. If no exception handler is found, the exception flag remains set and the machine halts. If this procedure does find an exception handler (f, t, C, h, d) , it cuts down the operand stack of the frame to d elements, puts a reference to the exception on top, and sets the *pc* to h . This is different from the Java VM where the stack is always emptied. Thus exception handling in the Jinja VM is a generalisation of the Java VM where d is always 0. Leaving a number of elements on the stack gives us a nice way to translate **try-catch** constructs of the source language that handle exceptions in the middle of an expression as opposed to just on the statement level as in Java.

For some proofs the relational view is more convenient than the functional one. Therefore we also define the one-step state transition relation:

$$P \vdash \sigma \xrightarrow{\text{jvm}}_1 \sigma' = (\text{exec } P \ \sigma = \lfloor \sigma' \rfloor)$$

The state transition relation $\xrightarrow{\text{jvm}}$ is the reflexive transitive closure of $\xrightarrow{\text{jvm}}_1$.

The definition of *exec-instr* in Fig. 18 is large, but straightforward. The parameters of *exec-instr* are the following: the instruction to execute, the program P , the heap h , the operand stack *stk* and local variables *loc* of the current call frame, the class C_0 and name M_0 of the method that is currently executed, the current *pc*, and the rest of the call frame stack *frs*. One of the smaller definitions in *exec-instr* is the one for the **IAdd** instruction:

$$\text{exec-instr } \mathbf{IAdd} \ P \ h \ (\text{Intg } i_2 \cdot \text{Intg } i_1 \cdot \text{stk}) \ \text{loc} \ C_0 \ M_0 \ \text{pc} \ \text{frs} = \\ (\text{None}, h, (\text{Intg } (i_1 + i_2) \cdot \text{stk}, \text{loc}, C_0, M_0, \text{pc} + 1) \cdot \text{frs})$$

It takes the top two values as integers from the stack, adds them and puts the result back onto the stack. The program counter is incremented, the rest remains untouched.

Most instructions in Fig. 18 are of this simple form. Some of them use new functions: *hd* and *tl* return the head and tail of a list, and the destructor *the-Addr* is defined by the equality *the-Addr* (*Addr* a) = a . The **New** instruction needs *new-Addr* which was introduced in §2.2. A new object of class C with all fields set to default values is produced by *blank* $P \ C$ (not shown). Remember that the field part of objects is a map from name and defining class to value, so *fs* (F, C) used for **Getfield** and **Putfield** is the value of field F defined in class C . The **Checkcast** instruction uses *cast-ok* $P \ C \ h \ v$ (also not shown) to check if the value v is an address that points to an object of at least class C .

The definition for **Invoke** $M \ n$ is the most complex: it first uses *take* $n \ \text{stk}$ to get the the first n elements of the stack (the parameters in reverse order), then it looks up the dynamic class C of the object, determines the correct method (using *method* $P \ C \ M$), and finally constructs the new state. If the object reference r is *Null* an exception is thrown, otherwise a new call frame for the invoked method is prepared. The new call frame has an empty operand stack, the object reference r as *this* pointer in local variable 0, the parameters (*rev ps* is ps in reverse order) in the next n variables, and the rest of the local variables filled with a dummy value

```

exec-instr (Load n) P h stk loc C0 M0 pc frs = (None, h, (loc[n] · stk, loc, C0, M0, pc + 1) · frs)
exec-instr (Store n) P h stk loc C0 M0 pc frs =
  (None, h, (tl stk, loc[n := hd stk], C0, M0, pc + 1) · frs)
exec-instr (Push v) P h stk loc C0 M0 pc frs = (None, h, (v · stk, loc, C0, M0, pc + 1) · frs)
exec-instr (New C) P h stk loc C0 M0 pc frs =
  (case new-Addr h of None ⇒ ([addr-of-sys-xcpt OutOfMemory], h, (stk, loc, C0, M0, pc) · frs)
   | [a] ⇒ (None, h(a ↦ blank P C), (Addr a · stk, loc, C0, M0, pc + 1) · frs))
exec-instr (Getfield F C) P h stk loc C0 M0 pc frs =
  (let v · _ = stk; xp' = if v = Null then [addr-of-sys-xcpt NullPointer] else None;
   [ (D, fs) ] = h (the-Addr v)
   in (xp', h, (the (fs (F, C)) · tl stk, loc, C0, M0, pc + 1) · frs))
exec-instr (Putfield F C) P h stk loc C0 M0 pc frs =
  (let v · _ = stk; _ · r · _ = stk; xp' = if r = Null then [addr-of-sys-xcpt NullPointer] else None;
   Addr a = r; [ (D, fs) ] = h a; h' = h(a ↦ (D, fs((F, C) ↦ v)))
   in (xp', h', (tl (tl stk), loc, C0, M0, pc + 1) · frs))
exec-instr (Checkcast C) P h stk loc C0 M0 pc frs =
  (let v · _ = stk; xp' = if ¬ cast-ok P C h v then [addr-of-sys-xcpt ClassCast] else None
   in (xp', h, (stk, loc, C0, M0, pc + 1) · frs))
exec-instr (Invoke M n) P h stk loc C0 M0 pc frs =
  (let ps = take n stk; r = stk[n];
   xp' = if r = Null then [addr-of-sys-xcpt NullPointer] else None;
   [ (C, _) ] = h (the-Addr r); (D, M', Ts, mxs, mxl0, ins, xt) = method P C M;
   f' = ([], [r] @ rev ps @ replicate mxl0 arbitrary, D, M, 0)
   in (xp', h, f' · (stk, loc, C0, M0, pc) · frs))
exec-instr Return P h stk0 loc0 C0 M0 pc frs =
  (if frs = [] then (None, h, [])
   else let v · _ = stk0; (stk, loc, C, m, pc) · _ = frs; n = |fst (snd (method P C0 M0))|
   in (None, h, (v · drop (n + 1) stk, loc, C, m, pc + 1) · tl frs))
exec-instr Pop P h stk loc C0 M0 pc frs = (None, h, (tl stk, loc, C0, M0, pc + 1) · frs)
exec-instr IAdd P h stk loc C0 M0 pc frs =
  (let Intg i2 · _ = stk; _ · Intg i1 · _ = stk
   in (None, h, (Intg (i1 + i2) · tl (tl stk), loc, C0, M0, pc + 1) · frs))
exec-instr (IfFalse i) P h stk loc C0 M0 pc frs =
  (let pc' = if hd stk = Bool False then nat (int pc + i) else pc + 1
   in (None, h, (tl stk, loc, C0, M0, pc') · frs))
exec-instr CmpEq P h stk loc C0 M0 pc frs =
  (let v2 · _ = stk; _ · v1 · _ = stk
   in (None, h, (Bool (v1 = v2) · tl (tl stk), loc, C0, M0, pc + 1) · frs))
exec-instr (Goto i) P h stk loc C0 M0 pc frs =
  (None, h, (stk, loc, C0, M0, nat (int pc + i)) · frs)
exec-instr Throw P h stk loc C0 M0 pc frs =
  (let xp' = if hd stk = Null then [addr-of-sys-xcpt NullPointer] else [the-Addr (hd stk)]
   in (xp', h, (stk, loc, C0, M0, pc) · frs))

```

Fig. 18. Single step execution

arbitrary (*replicate* n v is a list of length n that contains only v -elements). The new call frame also records the method (defining class D and name M), and has the pc set to 0.

It is in this **Invoke** context that **Return** is understood best: if the current frame is the only one on the frame stack, the machine halts; if the current frame is not the only one, then the next frame on the stack must be the caller where the corresponding **Invoke** occurred. The **Return** instruction removes the current frame from the call stack and manipulates the caller frame. In the caller, it drops the parameters and the object reference, i.e. $n + 1$ elements, from the stack (*drop* n xs is the dual to *take* n xs), puts the return value v on top, and increments the pc .

This style of VM is also called *aggressive*, because it does not perform any runtime type or sanity checks. It just assumes that everything is as expected, e.g. for **IAdd** that there are indeed two integers on the stack. If the situation is not as expected, the operational semantics is unspecified at this point. In Isabelle, this means that there is a result (because HOL is a logic of total functions), but nothing is known about that result. It is the task of the bytecode verifier to ensure that this does not occur.

3.3 A Defensive VM

Although it is possible to prove type safety by using the aggressive VM alone, it is crisper to write and more obvious to see what the bytecode verifier guarantees when we additionally look at a defensive VM. The defensive VM builds on the aggressive one by performing extra type and sanity checks. We can then state the type safety theorem by saying that these checks will never fail if the bytecode is well-typed. This differs from the approach of Wright and Felleisen [1994] that we follow for the source language where we prove progress and preservation instead. The separation into *exec-instr* and *check-instr* allows us to write down the semantics in a functional rather than relational style at the single-step level. This in turn results in a higher degree of automation in the proofs.

To indicate type errors, we introduce another data type.

$$\text{datatype } 'a \text{ type-error} = \text{TypeError} \mid \text{Normal } 'a$$

Similar to §3.2, we build on a function *check-instr* that is lifted over several steps. At the deepest level, we take apart the state, check if the current method exists, feed *check-instr* with parameters (which are the same as for *exec-instr*), and check that pc and stack size are valid:

$$\begin{aligned} \text{check } P \sigma &\equiv \\ \text{let } (xcpt, h, frs) &= \sigma \\ \text{in case frs of } [] &\Rightarrow \text{True} \\ &\mid (stk, loc, C, M, pc) \cdot frs' \Rightarrow \\ &\quad P \vdash C \text{ has } M \wedge \\ &\quad (\text{let } (C', Ts, T, mxs, mxl_0, ins, xt) = \text{method } P \ C \ M; i = \text{ins}_{[pc]} \\ &\quad \text{in } pc < |ins| \wedge |stk| \leq mxs \wedge \text{check-instr } i \ P \ h \ stk \ loc \ C \ M \ pc \ frs') \end{aligned}$$

The next level is the one-step execution of the defensive VM, which stops in case of a type error. If there is no type or other error, it simply calls the aggressive VM:

$$\text{exec}_d \ P \ \sigma \equiv \text{if check } P \ \sigma \ \text{then Normal } (\text{exec } P \ \sigma) \ \text{else TypeError}$$

$check_instr$ (**Load** n) $P h stk loc C M_0 pc frs = (n < |loc|)$
 $check_instr$ (**Store** n) $P h stk loc C_0 M_0 pc frs = (0 < |stk| \wedge n < |loc|)$
 $check_instr$ (**Push** v) $P h stk loc C_0 M_0 pc frs = (\neg is_Addr v)$
 $check_instr$ (**New** C) $P h stk loc C_0 M_0 pc frs = is_class P C$
 $check_instr$ (**Getfield** $F C$) $P h stk loc C_0 M_0 pc frs =$
 $(0 < |stk| \wedge (\exists C' T. P \vdash C \text{ sees } F:T \text{ in } C') \wedge$
 $(let (C', T) = field P C F; ref \cdot _ = stk$
 $in C' = C \wedge is_Ref ref \wedge$
 $(ref \neq Null \longrightarrow$
 $h (the_Addr ref) \neq None \wedge$
 $(let [(D, vs)] = h (the_Addr ref)$
 $in P \vdash D \preceq^* C \wedge vs (F, C) \neq None \wedge P, h \vdash the (vs (F, C)) : \leq T))))$
 $check_instr$ (**Putfield** $F C$) $P h stk loc C_0 M_0 pc frs =$
 $(1 < |stk| \wedge (\exists C' T. P \vdash C \text{ sees } F:T \text{ in } C') \wedge$
 $(let (C', T) = field P C F; v \cdot _ = stk; \cdot \cdot ref \cdot _ = stk$
 $in C' = C \wedge is_Ref ref \wedge$
 $(ref \neq Null \longrightarrow$
 $h (the_Addr ref) \neq None \wedge$
 $(let [(D, _)] = h (the_Addr ref) in P \vdash D \preceq^* C \wedge P, h \vdash v : \leq T))))$
 $check_instr$ (**Checkcast** C) $P h stk loc C_0 M_0 pc frs =$
 $(0 < |stk| \wedge is_class P C \wedge is_Ref (hd stk))$
 $check_instr$ (**Invoke** $M n$) $P h stk loc C_0 M_0 pc frs =$
 $(n < |stk| \wedge is_Ref stk_{[n]} \wedge$
 $(stk_{[n]} \neq Null \longrightarrow$
 $(let Addr a = stk_{[n]}; [(C, _)] = h a; (\cdot, Ts, \cdot) = method P C M$
 $in h a \neq None \wedge P \vdash C \text{ has } M \wedge P, h \vdash rev (take n stk) [:\leq Ts]))$
 $check_instr$ **Return** $P h stk loc C_0 M_0 pc frs =$
 $(0 < |stk| \wedge$
 $(0 < |frs| \longrightarrow$
 $P \vdash C_0 \text{ has } M_0 \wedge (let v \cdot _ = stk; (\cdot, \cdot, T, \cdot) = method P C_0 M_0 in P, h \vdash v : \leq T))))$
 $check_instr$ **Pop** $P h stk loc C_0 M_0 pc frs = (0 < |stk|)$
 $check_instr$ **IAdd** $P h stk loc C_0 M_0 pc frs =$
 $(1 < |stk| \wedge is_Intg (hd stk) \wedge is_Intg (hd (tl stk)))$
 $check_instr$ (**IfFalse** b) $P h stk loc C_0 M_0 pc frs =$
 $(0 < |stk| \wedge is_Bool (hd stk) \wedge 0 \leq int pc + b)$
 $check_instr$ **CmpEq** $P h stk loc C_0 M_0 pc frs = (1 < |stk|)$
 $check_instr$ (**Goto** b) $P h stk loc C_0 M_0 pc frs = (0 \leq int pc + b)$
 $check_instr$ **Throw** $P h stk loc C_0 M_0 pc frs = (0 < |stk| \wedge is_Ref (hd stk))$

Fig. 19. Type checks in the defensive Jinja VM.

Again, we also define the relational view. This time, it is easier to give two introduction rules for $\xrightarrow{d_{jvm}}_1$:

$$\frac{exec_d P \sigma = TypeError}{P \vdash Normal \sigma \xrightarrow{d_{jvm}}_1 TypeError} \quad \frac{exec_d P \sigma = Normal [\sigma']}{P \vdash Normal \sigma \xrightarrow{d_{jvm}}_1 Normal \sigma'}$$

We write $\xrightarrow{d_{jvm}}$ for the reflexive transitive closure of $\xrightarrow{d_{jvm}}_1$.

It remains to define *check-instr*, the heart of the defensive Jinja VM. We do so in Fig. 19. The **IAdd** case looks like this:

$$\text{check-instr } \mathbf{IAdd} \ P \ h \ stk \ loc \ C_0 \ M_0 \ pc \ frs = (1 < |stk| \wedge is-Intg (hd \ stk) \wedge is-Intg (hd (tl \ stk)))$$

IAdd requires that the stack has at least two entries ($1 < |stk|$), and that these entries are of type *Integer* (checked with the *is-Intg* function). For **Load** and **Store** there are no type constraints, because they are polymorphic in Jinja. In the Java VM, the definition would be in the style of **IAdd**, requiring integer for **iload**, float for **fload**, and so on. The discriminator functions *is-Addr* and *is-Ref* in Fig. 19 do the obvious.

Because of its size, we also take a closer look at the instruction **Getfield**:

$$\begin{aligned} \text{check-instr } (\mathbf{Getfield} \ F \ C) \ P \ h \ stk \ loc \ C_0 \ M_0 \ pc \ frs = \\ (0 < |stk| \wedge (\exists C' \ T. P \vdash C \ \text{sees} \ F:T \ \text{in} \ C') \wedge \\ (\text{let } (C', T) = \text{field} \ P \ C \ F; \ \text{ref} \cdot _ = \text{stk} \\ \text{in } C' = C \wedge is-Ref \ \text{ref} \wedge \\ (\text{ref} \neq \text{Null} \longrightarrow \\ h \ (\text{the-Addr} \ \text{ref}) \neq \text{None} \wedge \\ (\text{let } _ (D, vs) = h \ (\text{the-Addr} \ \text{ref}) \\ \text{in } P \vdash D \preceq^* C \wedge vs \ (F, C) \neq \text{None} \wedge P, h \vdash \text{the} \ (vs \ (F, C)) \preceq T)))) \end{aligned}$$

The **Getfield** $F \ C$ instruction is supposed to access the object reference on top of the stack, remove it from the stack, and to put the value of field F defined in class C onto the stack instead. To ensure this can work without errors, the first two conjuncts in this definition demand that the stack is large enough to hold the object reference, and that the field F is visible from class C . The *let*-part collects the defining class and the type of field F , as well as the object reference. The next line checks that the field is of the right class ($C' = C$) and that we are indeed dealing with an object reference. If that reference does not happen to be *Null*, the heap at position *ref* must contain an object of class at least C , the object must contain a field F ($vs \ (F, C) \neq \text{None}$), and the field value must be of the type that was declared for the field. The **Putfield** instruction works analogously.

It is easy to see that defensive and aggressive VM have the same operational one-step semantics if there are no type errors.

THEOREM 3.1. One-step execution in aggressive and defensive machines commutes if there are no type errors.

$$\text{exec}_d \ P \ \sigma \neq \text{TypeError} \implies \text{exec}_d \ P \ \sigma = \text{Normal} \ (\text{exec} \ P \ \sigma)$$

Figure 20 depicts this result as a commuting diagram. The proof is trivial (and fully automatic in Isabelle), because the defensive VM is constructed directly from the aggressive one.

For executing programs, we will later also need a canonical start state. In the Java VM, a program is started by invoking its static *main* method. We start the Jinja VM by invoking any existing method M (without parameters) in a class C . We define the canonical start state $\text{start} \ P \ C \ M$ as the state with exception flag *None*, a heap $\text{start-heap} \ P$, and a frame stack with one element. The heap $\text{start-heap} \ P$ contains the preallocated system exceptions and is otherwise empty. The single frame has an empty operand stack, the *this* pointer set to *Null*, the rest of the register set filled up with a dummy value *arbitrary*, the class entry set to C , the name to M , and the program counter 0.

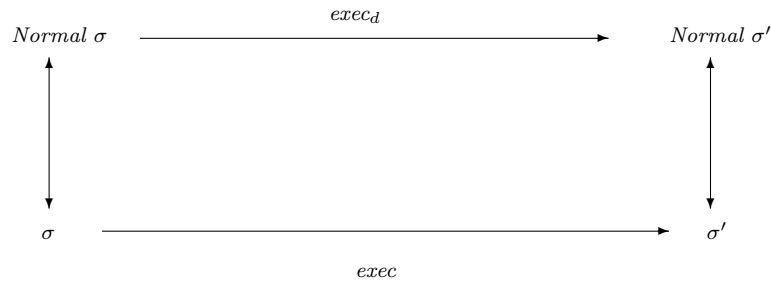


Fig. 20. Aggressive and defensive Jinja VM commute if there are no type errors.

```

start-state P C M ≡
let (D, Ts, T, mxs, mxl0, b) = method P C M
in (None, start-heap P, [([] , Null · replicate mxl0 arbitrary, C, M, 0)])

```

This concludes the formalisation of the Jinja VM. It will serve as the basis for the proof of type safety below.

3.4 Related Work

Most closely related to the virtual machine formalisation presented here is the Ph.D. thesis by Klein [2003]. Barthe et al. [2001] use the Coq system for a similar formalisation of the Java VM. Liu and Moore [2003] show one of the most comprehensive executable models of the KVM (a restricted JVM for embedded devices) in the theorem prover ACL2. Cohen [1997] was the first to propose the concept of a defensive Java VM and to formalise it in a theorem prover. Bertelsen [1997] was one of the first to give a formal semantics for the JVM. Most publications about Java’s bytecode verifier contain some form of reference virtual machine, be that in the form of a traditional small step semantics, or in an explicitly executable format in a theorem prover. Instead of recounting all of these here we refer to the overview articles [Alves-Foss 1999; Hartel and Moreau 2001; Nipkow 2003a] and the related work section on bytecode verification in §4.11.

4. BYTECODE VERIFIER

The JVM relies on the following assumptions for executing bytecode:

Correct types. All bytecode instructions are provided with arguments of the type they expect on operand stack, registers, and heap.

No overflow and underflow. No instruction tries to retrieve a value from the empty stack, no instruction puts more elements on the stack than statically specified in the method, and no instruction accesses more registers than statically specified in the method.

Code containment. The program counter is always within the code array of the method. Specifically, it must not fall off the end of the method’s code.

Initialised registers. All registers apart from the *this* pointer and the method parameters must be written to before they are first read. This corresponds to the *definite assignment* requirement for local variables on the source level.

instruction	stack	registers
<i>Load</i> 0	[([], [Class B, Integer])]	
<i>Store</i> 1	[([Class A], [Class B, Err])]	
<i>Load</i> 0	[([], [Class B, Class A])]	
<i>Getfield</i> F A	[([Class B], [Class B, Class A])]	
<i>Goto</i> -3	[([Class A], [Class B, Class A])]	

Fig. 21. Example of a method well-typing.

It is the purpose of the bytecode verifier (BV) to ensure statically that these assumptions are met at any time during execution.

Bytecode verification is an abstract interpretation of bytecode methods: instead of values, we only consider their types. A **state type** characterises a set of runtime states by giving type information for the operand stack and registers. For example, the first state type in Fig. 21 ($([], [Class B, Integer])$) characterises all states whose stack is empty, whose register 0 contains a reference to an object of class B (or to a subclass of B), and whose register 1 contains an integer. A method is called **well-typed** if we can assign a well-typing to each instruction. A state type (ST, LT) is a well-typing for an instruction if it is consistent with the successors of the instruction and if the instruction can be executed safely in any state whose stack is typed according to ST and whose registers are typed according to LT . In other words: the arguments of the instruction are provided in correct number, order and type. We explain *consistent* below.

The example in Fig. 21 shows the instructions on the left and the type of stack elements and registers on the right. The **method type** is the full right-hand side of the table, a state type is one line of it. The type information attached to an instruction characterises the state *before* execution of that instruction. The [...] around each entry means that it was possible to predict some type for each instruction. If one of the instructions had been unreachable, the type entry would have been *None*. We assume that class B is a subclass of A and that A has a field F of type A .

Execution starts with an empty stack and the two registers holding a reference to an object of class B and an integer. The first instruction loads register 0, a reference to a B object, on the stack. The type information associated with the following instruction may puzzle at first sight: it says that a reference to an A object is on the stack, and that usage of register 1 may produce an error. This means the type information has become less precise but is still correct: a B object is also an A object and an integer is now classified as unusable (*Err*). The reason for these more general types is that the predecessor of the *Store* instruction may have either been *Load* 0 or *Goto* -3. Since there exist different execution paths to reach *Store*, the type information of the two paths has to be *merged*. The type of the second register is either *Integer* or *Class A*, which are incompatible: the only common supertype is *Err*.

Bytecode verification is the process of inferring the types on the right from the instruction sequence on the left and some initial condition, and of ensuring that each instruction receives arguments of the correct type. Type inference is the computation of a method type from an instruction sequence, type checking means

checking that a given method type fits an instruction sequence.

Figure 21 was an example for a well-typed method: we were able to find a well-typing. If we changed the third instruction from *Load 0* to *Store 0*, the method would not be well-typed. The *Store* instruction would try to take an element from the empty stack and could therefore not be executed.

In the following, we will formalise the Jinja bytecode verifier by introducing an abstract framework for well-typedness, then instantiating this framework in §4.6 and §4.7 to get a description of well-typedness for Jinja VM programs as well as an executable program that computes those well-typings in §4.8 and §4.9. Finally we show in §4.10 that execution of well-typed programs is safe in the sense motivated above.

The typing framework uses semilattices (§4.1) and an abstract transfer function to describe well-typings (§4.2). After detailing which constraints on the transfer function are necessary to obtain an executable algorithm in §4.3 and refining the transfer function to make instantiation easier in §4.4, we show an implementation of Kildall’s algorithm within the framework (§4.5). As the framework itself already appeared elsewhere [Klein 2003; Klein and Nipkow 2003; Nipkow 2001], we will keep its description brief and only reproduce the main definitions and properties.

4.1 Semilattices

This section introduces the formalisation of the basic lattice-theoretic concepts required for data flow analysis and its application to the JVM.

4.1.1 Partial Orders. Partial orders are formalised as binary predicates. Based on the type synonym $'a \text{ ord} = 'a \Rightarrow 'a \Rightarrow \text{bool}$ and the notations $x \sqsubseteq_r y \equiv r \ x \ y$ and $x \sqsubset_r y \equiv x \sqsubseteq_r y \wedge x \neq y$, $r :: 'a \text{ ord}$ is by definition a **partial order** iff the predicate $\text{order } r :: 'a \text{ ord} \Rightarrow \text{bool}$ holds for r :

$$\begin{aligned} \text{order } r \equiv & \\ (\forall x. x \sqsubseteq_r x) \wedge & (\forall x \ y. x \sqsubseteq_r y \wedge y \sqsubseteq_r x \longrightarrow x = y) \wedge (\forall x \ y \ z. x \sqsubseteq_r y \wedge y \sqsubseteq_r z \longrightarrow x \sqsubseteq_r z) \end{aligned}$$

A partial order r satisfies the **ascending chain condition** on A if there is no infinite ascending chain $x_0 \sqsubset_r x_1 \sqsubset_r \dots$ in A , and \top is called a **top element** if $x \sqsubseteq_r \top$ for all x . Instead of “no infinite ascending chain” we require in Isabelle the equivalent “the converse of r is well-founded”.

$$\text{acc } r \equiv \text{wf } \{(y, x) \mid x \sqsubset_r y\} \quad \text{top } r \ \top \equiv \forall x. x \sqsubseteq_r \top$$

4.1.2 Semilattices. Based on the supremum notation $x \sqcup_f y \equiv f \ x \ y$ and the two type synonyms $'a \text{ binop} = 'a \Rightarrow 'a \Rightarrow 'a$ and $'a \text{ sl} = 'a \text{ set} \times 'a \text{ ord} \times 'a \text{ binop}$, the tuple $(A, r, f) :: 'a \text{ sl}$ is by definition a **semilattice** iff the predicate $\text{semilat} :: 'a \text{ sl} \Rightarrow \text{bool}$ holds:

$$\begin{aligned} \text{semilat } (A, r, f) \equiv & \\ \text{order } r \wedge \text{closed } A \ f \wedge & (\forall x \in A. \forall y \in A. x \sqsubseteq_r x \sqcup_f y) \wedge (\forall x \in A. \forall y \in A. y \sqsubseteq_r x \sqcup_f y) \wedge \\ (\forall x \in A. \forall y \in A. \forall z \in A. & x \sqsubseteq_r z \wedge y \sqsubseteq_r z \longrightarrow x \sqcup_f y \sqsubseteq_r z) \end{aligned}$$

where $\text{closed } A \ f \equiv \forall x \in A. \forall y \in A. x \sqcup_f y \in A$.

Data flow analysis is usually phrased in terms of infimum semilattices. Here, a supremum semilattice fits better with the intended application, where the ordering is the subtype relation and the join of two types is the least common supertype (if it exists).

The next sections look at a few data types and the corresponding semilattices which are required for the construction of the Jinja VM bytecode verifier. The definition of those semilattices follows a pattern: they lift an existing semilattice to a new semilattice with more structure. They extend the carrier set and define two functionals le and sup that lift the ordering and supremum operation to the new semilattice. In order to avoid name clashes, Isabelle provides separate name spaces for each theory. Qualified names are of the form $Theoryname.localname$, and they apply to constant definitions and functions as well as type constructions. So $Err.sup$ later on refers to the sup functional defined for the error type in §4.1.3.

Let (A, r, f) in the following be a semilattice.

4.1.3 The Error Type and Err-semilattices. Theory Err introduces an error element to model the situation where the supremum of two elements does not exist. It introduces both a data type and an equivalent construction on sets:

datatype $'a\ err = Err \mid OK\ 'a$ $err\ A \equiv \{Err\} \cup \{OK\ x \mid x. x \in A\}$

The additional x in $\{OK\ x \mid x. x \in A\}$ tells Isabelle that x is a bound variable. It is used if the set comprehension is of the form $\{f\ x \mid x. P\ x\}$ rather than just $\{x \mid P\ x\}$. An ordering r on $'a$ can be lifted to $'a\ err$ by making Err the top element:

$$\begin{aligned} le\ r\ (OK\ x)\ (OK\ y) &= x \sqsubseteq_r y \\ le\ r\ _\ Err &= True \\ le\ r\ Err\ (OK\ _) &= False \end{aligned}$$

LEMMA 4.1. If $acc\ r$ then $acc\ (le\ r)$.

The following lifting functional is frequently useful:

$$\begin{aligned} lift_2\ f\ (OK\ x)\ (OK\ y) &= f\ x\ y \\ lift_2\ f\ Err\ _ &= Err \\ lift_2\ f\ _\ Err &= Err \end{aligned}$$

This leads to the notion of an err-semilattice. It is a variation of a semilattice with top element. Because the behaviour of the ordering and the supremum on the top element is fixed, it suffices to say how ordering and supremum behave on non-top elements. Thus we can represent a semilattice with top element Err compactly by a triple of type esl :

$$'a\ ebinop = 'a \Rightarrow 'a \Rightarrow 'a\ err \quad 'a\ esl = 'a\ set \times 'a\ ord \times 'a\ ebinop$$

Conversion between the types sl and esl is easy:

$$\begin{aligned} esl :: 'a\ sl \Rightarrow 'a\ esl & & sl :: 'a\ esl \Rightarrow 'a\ err\ sl \\ esl\ (A, r, f) \equiv (A, r, \lambda x\ y. OK\ (f\ x\ y)) & & sl\ (A, r, f) \equiv (err\ A, le\ r, lift_2\ f) \end{aligned}$$

A tuple $L :: 'a\ esl$ is by definition an **err-semilattice** iff $sl\ L$ is a semilattice. Conversely, we have Lemma 4.2.

LEMMA 4.2. If $semilat\ L$ then $err-semilat\ (esl\ L)$.

The supremum operation of $sl(esl\ L)$ is useful on its own:

$$sup\ f \equiv lift_2\ (\lambda x\ y. OK\ (x \sqcup_f y))$$

4.1.4 *The Option Type.* Theory *Opt* uses the type *option* and introduces the set *opt* as dual to set *err*,

$$\text{opt } A \equiv \{\text{None}\} \cup \{[y] \mid y. y \in A\}$$

an ordering that makes *None* the bottom element, and a corresponding supremum operation:

$$\begin{aligned} \text{le } r \ [x] \ [y] &= x \sqsubseteq_r y \\ \text{le } r \ \text{None} \ _ &= \text{True} \\ \text{le } r \ _ \ \text{None} &= \text{False} \\ \text{sup } f \ [x] \ [y] &= (\text{case } f \ x \ y \ \text{of } \text{Err} \Rightarrow \text{Err} \mid \text{OK } z \Rightarrow \text{OK } [z]) \\ \text{sup } f \ z \ \text{None} &= \text{OK } z \\ \text{sup } f \ \text{None} \ z &= \text{OK } z \end{aligned}$$

LEMMA 4.3. Let $\text{esl } (A, r, f) = (\text{opt } A, \text{le } r, \text{sup } f)$. If *err-semilat* *L* then *err-semilat* ($\text{esl } L$).

It is possible to define an *sl* that lifts a semilattice to an option semilattice, but we only use the *esl* version below.

LEMMA 4.4. If *acc* *r* then *acc* (*le* *r*).

4.1.5 *Products.* Theory *Product* provides what is known as the **coalesced** product, where the top elements of both components are identified. In terms of *err-semilattices*, this is:

$$\begin{aligned} \text{esl} &:: 'a \ \text{esl} \Rightarrow 'b \ \text{esl} \Rightarrow ('a \times 'b) \ \text{esl} \\ \text{esl } (A, r_A, f_A) \ (B, r_B, f_B) &\equiv (A \times B, \text{le } r_A \ r_B, \text{sup } f_A \ f_B) \\ \text{le} &:: 'a \ \text{ord} \Rightarrow 'b \ \text{ord} \Rightarrow ('a \times 'b) \ \text{ord} \\ \text{le } r_A \ r_B \ (a_1, b_1) \ (a_2, b_2) &\equiv a_1 \sqsubseteq_{r_A} a_2 \wedge b_1 \sqsubseteq_{r_B} b_2 \\ \text{sup} &:: 'a \ \text{ebinop} \Rightarrow 'b \ \text{ebinop} \Rightarrow ('a \times 'b) \ \text{ebinop} \\ \text{sup } f \ g \ (a_1, b_1) \ (a_2, b_2) &\equiv \text{Err}.\text{sup } (\lambda x \ y. (x, y)) \ (a_1 \sqcup_f a_2) \ (b_1 \sqcup_g b_2) \end{aligned}$$

Note that \times is used both on the type and the set level.

LEMMA 4.5. If *err-semilat* *L*₁ and *err-semilat* *L*₂ then *err-semilat* ($\text{esl } L_1 \ L_2$).

LEMMA 4.6. If *acc* *r*_A and *acc* *r*_B then *acc* (*le* *r*_A *r*_B).

4.1.6 *Lists of Fixed Length.* Theory *Listn* provides the concept of lists of a given length over a given set. In HOL, this is formalised as a set rather than a type:

$$\text{list } n \ A \equiv \{xs \mid |xs| = n \wedge \text{set } xs \subseteq A\}$$

This set can be turned into a semilattice in a componentwise manner, essentially viewing it as an *n*-fold Cartesian product:

$$\begin{aligned} \text{sl} &:: \text{nat} \Rightarrow 'a \ \text{sl} \Rightarrow 'a \ \text{list} \ \text{sl} & \text{le} &:: 'a \ \text{ord} \Rightarrow 'a \ \text{list} \ \text{ord} \\ \text{sl } n \ (A, r, f) &\equiv (\text{list } n \ A, \text{le } r, \text{map}_2 \ f) & \text{le } r &\equiv \text{list-all}_2 \ (\lambda x \ y. x \sqsubseteq_r y) \end{aligned}$$

where $\text{map}_2 :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \ \text{list} \Rightarrow 'b \ \text{list} \Rightarrow 'c \ \text{list}$ and $\text{list-all}_2 :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \ \text{list} \Rightarrow 'b \ \text{list} \Rightarrow \text{bool}$ are the obvious functions. Below, we use the notation $xs \sqsubseteq_r ys$ for $xs \sqsubseteq_{\text{le } r} ys$.

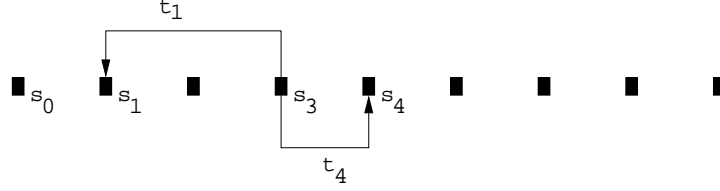


Fig. 22. Data flow graph for *step 3* $s_3 = [(1,t_1),(4,t_4)]$.

LEMMA 4.7. If *semilat* L then *semilat* $(sl\ n\ L)$.

LEMMA 4.8. If *order* r and *acc* r then *acc* $(le\ r)$.

In case we want to combine lists of different lengths, or if the supremum on the elements of the list may return *Err* (not to be confused with *Err.sup* the *sup* functional defined in Theory *Err*, §4.1.3), the following function is useful:

$$\begin{aligned} sup &:: ('a \Rightarrow 'b \Rightarrow 'c\ err) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list\ err \\ sup\ f\ xs\ ys &\equiv \text{if } |xs| = |ys| \text{ then } coalesce\ (map_2\ f\ xs\ ys) \text{ else } Err \\ coalesce\ [] &= OK\ [] \\ coalesce\ (e \cdot es) &= Err.sup\ (\lambda x\ xs.\ x \cdot xs)\ e\ (coalesce\ es) \end{aligned}$$

This corresponds to the coalesced product. Below, we also need the structure of all lists up to a specific length:

$$\begin{aligned} upto\ esl &:: nat \Rightarrow 'a\ esl \Rightarrow 'a\ list\ esl \\ upto\ esl\ n\ (A, r, f) &\equiv (\bigcup_{i \leq n} list\ i\ A, le\ r, sup\ f) \end{aligned}$$

LEMMA 4.9. If *err-semilat* L then *err-semilat* $(upto\ esl\ m\ L)$.

4.2 Well-Typings

This section describes well-typings abstractly. On this abstract level, there is no need yet to talk about the instruction sequences themselves. They will be hidden inside a function that characterises their behaviour. This function and a semilattice form the parameters of the model.

Data flow analysis and type systems are based on an abstract view of the semantics of a program in terms of types instead of values. At this level, programs are sequences of instructions, and the semantics can be characterised by a function $step :: pc \Rightarrow 's \Rightarrow (pc \times 's)\ list$ where pc is an abbreviation for nat . It is the abstract execution function: $step\ p\ s$ provides the results of executing the instruction at p , starting in state s , together with the positions where execution continues. Contrary to the usual concept of *transfer function* or *flow function* in the literature, $step\ p$ not only provides the result, but also the structure of the data flow graph at position p . This is best explained through an example. Figure 22 depicts the information we get when *step 3* s_3 returns the list $[(1,t_1),(4,t_4)]$: executing the instruction at position 3 with state type s_3 may lead to position 1 in the graph with result t_1 , or to position 4 with result t_4 .

Note that the length of the list and the target instructions do not only depend on the source position p in the graph, but also on the value of s . It is possible that the structure of the data flow graph dynamically changes in the iteration process of the

analysis. It may not change freely, however. §4.3 will introduce certain constraints on the *step* function that the analysis needs in order to succeed.

Data flow analysis is concerned with solving data flow equations, which are systems of equations involving the flow functions over a semilattice. In this case, *step* is the flow function and *'s* the semilattice. Instead of an explicit formalisation of the data flow equation, it suffices to consider certain prefixed points. To that end we define what it means that a method type $\tau s :: 's \text{ list}$ is **stable at** p :

$$\text{stable } \tau s \ p \equiv \forall (q, \tau) \in \text{set } (\text{step } p \ \tau s_{[p]}). \ \tau \sqsubseteq_r \tau s_{[q]}$$

Stability induces the notion of a method type τs being a **well-typing w.r.t.** *step*:

$$\text{wt-step } \tau s \equiv \forall p < |\tau s|. \ \tau s_{[p]} \neq \top \wedge \text{stable } \tau s \ p$$

\top is assumed to be a special element in the state space (the top element of the ordering). It indicates a type error.

An instruction sequence is **well-typed** if there is a well-typing τs such that *wt-step* τs .

4.3 Constraints on the Transfer Function

This section defines constraints on the transfer function that the data flow algorithm in §4.5 needs to succeed.

The transfer function *step* is called **monotone up to** n iff the following holds:

$$\text{mono } n \equiv \forall \tau \in A. \ \forall p < n. \ \forall \tau'. \ \tau \sqsubseteq_r \tau' \longrightarrow \text{set } (\text{step } p \ \tau) \{\sqsubseteq_r\} \text{set } (\text{step } p \ \tau')$$

where

$$A \{\sqsubseteq_r\} B \equiv \forall (p, \tau) \in A. \ \exists \tau'. \ (p, \tau') \in B \wedge \tau \sqsubseteq_r \tau'$$

This means, if we increase the state type τ at a position p , the data flow graph may have more edges (but not less), and the result at each edge may increase (but not decrease).

If for all $p < n$ and all τ the position components of *step* $p \ \tau$ are less than n , then *step* is **bounded by** n . This expresses that, from below instruction n , instruction n and beyond are unreachable: control never leaves the list of instructions below n .

$$\text{bounded } n \equiv \forall p < n. \ \forall \tau. \ \forall (q, \tau') \in \text{set } (\text{step } p \ \tau). \ q < n$$

If for all $p < n$ and $\tau \in A$ the values that *step* $p \ \tau$ returns are again in A , then *step* **preserves** A **up to** n :

$$\text{preserves } n \equiv \forall \tau \in A. \ \forall p < n. \ \forall (q, \tau') \in \text{set } (\text{step } p \ \tau). \ \tau' \in A$$

4.4 Refining the Transfer Function

The single transfer function *step* of §4.2 is compact and convenient for describing the abstract typing framework. For a large instantiation, however, it carries too much information in one place to be modular and intuitive. We will therefore first refine *step* into a part *app* for applicability and a part *eff* for the effect of instructions, and then instantiate these parts in §4.7. Furthermore, the state space *'s* will be of the form *'t err* for a suitable type *'t*, in which case the error element \top is *Err* itself. Given $\text{app} :: pc \Rightarrow 't \Rightarrow \text{bool}$, $\text{eff} :: pc \Rightarrow 't \Rightarrow (pc \times 't) \text{ list}$, and $n :: \text{nat}$ (the size of the method type), *step* is defined as follows:

$$\begin{aligned}
\text{step } p \text{ } Err &= \text{error} \\
\text{step } p \text{ } (OK \ \tau) &= (\text{if } \text{app } p \ \tau \text{ then } \text{map-snd } OK \ (\text{eff } p \ \tau) \text{ else } \text{error}) \\
\text{error} &\equiv \text{map } (\lambda x. (x, Err)) [0..<n] \\
\text{map-snd } f &\equiv \text{map } (\lambda(x, y). (x, f y))
\end{aligned}$$

The function *error* is used to propagate the error element *Err* to every position in the method type.

If we take the semilattice (A, r, f) to be an err-semilattice and the order r to be of the form $le \ r'$, we can similarly refine the notion of a well-typing w.r.t. *step* to a well-typing w.r.t. *app* and *eff*:

$$\text{wt-app-eff } \tau s \equiv \forall p < |\tau s|. \text{app } p \ \tau s_{[p]} \wedge (\forall (q, \tau) \in \text{set } (\text{eff } p \ \tau s_{[p]}). \tau \sqsubseteq_{r'} \tau s_{[q]})$$

This is very natural: every instruction is applicable in its start state, and the effect is compatible with the state expected by all successor instructions.

Function *step* composed of *app* and *eff* as defined above has type $pc \Rightarrow 't \text{ err} \Rightarrow (pc \times 't \text{ err}) \text{ list}$. This is an instance of the type the stability predicates in §4.2 expect. If we furthermore set n to $|\tau s|$, we get the following lemma.

LEMMA 4.10. If the composed function *step* is bounded by $n = |\tau s|$, then *wt-app-eff* and *wt-step* coincide:

$$\text{bounded } |\tau s| \implies \text{wt-step } (\text{map } OK \ \tau s) = \text{wt-app-eff } \tau s$$

4.5 Kildall's Algorithm

A well-typing is a witness of well-typedness in the sense of stability. Now we turn to the problem of computing such a witness. This is precisely the task of a bytecode verifier: it computes a method type such that the absence of \top in the result means the method is well-typed. Formally, a function $bcv :: 's \text{ list} \Rightarrow 's \text{ list}$ is a **bytecode verifier** w.r.t. $n :: \text{nat}$ and $A :: 's \text{ set}$ iff

$$\forall \tau s_0 \in \text{list } n \ A. (\forall p < n. (bcv \ \tau s_0)_{[p]} \neq \top) = (\exists \tau s \in \text{list } n \ A. \tau s_0 \sqsubseteq_r \tau s \wedge \text{wt-step } \tau s)$$

The notation \sqsubseteq_r lifts \sqsubseteq_r to lists (see also §4.1.6), \top is the top element of the semilattice. In practise, *bcv* τs_0 itself will be the well-typing, and it will also be the least well-typing. However, it is simpler not to require this.

This section first defines and then verifies a functional version of Kildall's algorithm [Kildall 1973; Muchnick 1997], a standard data flow analysis tool. In fact, the description of bytecode verification in the official JVM specification [Lindholm and Yellin 1999, pages 129–130] is essentially Kildall's algorithm, an iterative computation of the solution to the data flow problem. The main loop operates on a method type τs and a **worklist** $w :: pc \text{ set}$. The worklist contains the indices of those elements of τs that have changed and whose changes still need to be propagated to their successors. Each iteration picks an element p from w , executes instruction number p , and propagates the new states to the successor instructions of p . Iteration terminates once w becomes empty: in each iteration, p is removed but new elements can be added to w . The algorithm is expressed in terms of a predefined *while*-combinator of type $('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$ which satisfies the recursion equation

$$\text{while } b \text{ c } s = (\text{if } b \text{ s then while } b \text{ c } (c \text{ s}) \text{ else } s)$$

The term $\text{while } (\lambda s. b \text{ s}) (\lambda s. c \text{ s})$ is the functional counterpart of the imperative program $\text{while } b(s) \text{ do } s := c(s)$. The main loop can now be expressed as

$$\begin{aligned} \text{iter } \tau s \ w &\equiv \\ \text{while } (\lambda(\tau s, w). w \neq \{\}) & \\ (\lambda(\tau s, w). \text{let } p = \text{SOME } p. p \in w \text{ in } & \text{propa } (\text{step } p \ \tau s_{[p]}) \ \tau s \ (w - \{p\})) \\ (\tau s, w) & \end{aligned}$$

Since the choice $\text{SOME } p. p \in w$ in iter is guarded by $w \neq \{\}$, we know that there is a $p \in w$. An implementation is free to choose whichever element it wants.

Propagating the results qs of executing instruction number p to all successors is expressed by the primitive recursive function propa :

$$\begin{aligned} \text{propa } [] \ \tau s \ w &= (\tau s, w) \\ \text{propa } (q'. qs) \ \tau s \ w &= \\ (\text{let } (q, \tau) = q'; u = \tau \sqcup_f \tau s_{[q]}; w' = & \text{if } u = \tau s_{[q]} \text{ then } w \text{ else } \{q\} \cup w \\ \text{in } \text{propa } qs \ (\tau s[q := u]) \ w') & \end{aligned}$$

In the terminology of the official JVM specification [Lindholm and Yellin 1999, page 130], τ is merged with the state of all successor instructions q , i.e., the supremum is computed. If this results in a change of $\tau s_{[q]}$, then q is inserted into w .

Kildall's algorithm is simply a call to iter where the worklist is initialised with the set of unstable indices; upon termination we project on the first component:

$$\text{kildall } \tau s \equiv \text{fst } (\text{iter } \tau s \ \{p \mid p < |\tau s| \wedge \neg \text{stable } \tau s \ p\})$$

The key theorem is that Kildall's algorithm is a bytecode verifier as defined above:

THEOREM 4.11. If (A, r, f) is a semilattice, r meets the ascending chain condition on A , and step is monotone, preserving, and bounded w.r.t. A and n , then kildall is a bytecode verifier w.r.t. A and n .

PROOF. The correctness proof proceeds along the following lines. (1) Given the assumptions of Theorem 4.11, kildall is a total function: The work list either becomes smaller, or, if new positions are introduced, the elements they point to are larger than the element at the position that was taken out. Since there are no infinitely ascending chains in r , the algorithm must terminate. (2) The following are invariants of the analysis (mainly because of monotonicity of step): all positions not in the worklist are stable, and the computed method type is always between the start value τs_0 and any well-typing τs which is stable everywhere and satisfies $\tau s_0 \sqsubseteq_r \tau s$. Upon termination the worklist is empty, and $\text{kildall } \tau s_0$ is stable everywhere. With (1) and (2), Theorem 4.11 follows easily. The implication from left to right holds with the result of the algorithm as witness. The result is stable, it does not contain \top (by assumption), and it is above τs_0 because of (2). The implication from right to left holds, because wt-step implies that τs is stable and does not contain \top . Because of (2), $\text{kildall } \tau s_0$ is smaller than τs , hence it does not contain \top either. \square

This specification of Kildall's algorithm is executable: the worklist (in the specification a set) can be implemented by a list, the SOME operator by hd .

4.6 Semilattice for the Jinja VM

This section takes the first step to instantiate the framework of §4.1 to §4.5. It defines the semilattice structure on which Jinja’s bytecode verifier builds. It begins by turning the Jinja types ty into a semilattice in theory $SemiType$ below. We can then use the abstract combinators of §4.1 to construct the stack and register structure.

The carrier set $types$ is easy: the set of all types declared in the program.

$$types\ P = \{T \mid is\text{-}type\ P\ T\}$$

The order is the standard subtype ordering \leq of Jinja and the supremum operation follows it.

$$\begin{aligned} sup &:: jvm\text{-}prog \Rightarrow ty \Rightarrow ty \Rightarrow ty\ err \\ sup\ P\ NT\ (Class\ C) &= OK\ (Class\ C) \\ sup\ P\ (Class\ C)\ NT &= OK\ (Class\ C) \\ sup\ P\ (Class\ C)\ (Class\ D) &= OK\ (Class\ (lub\ P\ C\ D)) \\ sup\ P\ t_1\ t_2 &= (if\ t_1 = t_2\ then\ OK\ t_1\ else\ Err) \end{aligned}$$

The lub function (not shown here) computes the least upper bound of two classes by walking up the class hierarchy until one is a subclass of the other. Since every class is a subclass of $Object$ in a well-formed program (see also $wf\text{-}prog$ in §2.4), this least upper bound is guaranteed to exist.

With $SemiType.esl\ P \equiv (types\ P, \lambda x\ y. P \vdash x \leq y, sup\ P)$ we have proved the following theorem.

THEOREM 4.12. If P is well-formed, then $SemiType.esl$ is an err-semilattice and the subtype ordering \leq satisfies the ascending chain condition:

$$\begin{aligned} wf\text{-}prog\ _ P &\Longrightarrow\ err\text{-}semilat\ (SemiType.esl\ P) \\ wf\text{-}prog\ _ P &\Longrightarrow\ acc\ (\lambda x\ y. P \vdash x \leq y) \end{aligned}$$

PROOF. The proof is easy: it is obvious that \leq is transitive and reflexive. If P is well-formed, \leq is also antisymmetric, hence a partial order. It satisfies the ascending chain condition, because, if P is well-formed, the class hierarchy is a tree with $Object$ at its top. We have already argued above that sup is well defined, and it is easy to see that it is closed w.r.t. $types\ P$. Hence $SemiType.esl\ P$ is an err-semilattice. \square

We can now construct the stack and register structure. State types in the Jinja BV are the same as in the example in Fig. 21: values on the operand stack must always contain a known type ty , values in the local variables may be of an unknown type and therefore be unusable (encoded by Err). To handle unreachable code, the BV needs an additional *option* layer: if $None$ occurs in the well-typing, the corresponding instruction is unreachable. On the HOL-type level:

$$\begin{aligned} \mathbf{types}\quad ty_s &= ty\ list & ty_l &= ty\ err\ list \\ ty_i &= ty_s \times ty_l & ty_i' &= ty_i\ option \end{aligned}$$

The types of the stack and the local variables are ty_s and ty_l . We call ty_i the **instruction type**, and let the term *state type* refer to either of ty_i or ty_i' . It will be clear from the context which of these is meant.

The data flow analysis also needs to indicate type errors during the algorithm (see also §4.4), so we arrive at ty_i' *err* for the semilattice construction.

Turning ty_i' *err* into a semilattice is easy, because all of its constituent types are (err-)semilattices. The expression stacks form a semilattice because the supremum of stacks of different size is *Err*; the local variables form a semilattice because their number *m_{xl}* is fixed:

$$\begin{aligned} stk\text{-}esl &:: jvm\text{-}prog \Rightarrow nat \Rightarrow ty_s\ esl \\ stk\text{-}esl\ P\ mxs &\equiv upto\text{-}esl\ mxs\ (SemiType.esl\ P) \\ \\ loc\text{-}sl &:: jvm\text{-}prog \Rightarrow nat \Rightarrow ty_l\ sl \\ loc\text{-}sl\ P\ mxl &\equiv Listn.sl\ mxl\ (Err.sl\ (SemiType.esl\ P)) \end{aligned}$$

Stack and local variables are combined in a coalesced product via *Product.esl* and then embedded into *option* and *err* to create the final semilattice for '*s* = ty_i' *err*':

$$\begin{aligned} sl &:: jvm\text{-}prog \Rightarrow nat \Rightarrow nat \Rightarrow ty_i'\ esl \\ sl\ P\ mxs\ mxl &\equiv Err.sl\ (Opt.esl\ (Product.esl\ (stk\text{-}esl\ P\ mxs)\ (Err.esl\ (loc\text{-}sl\ P\ mxl)))) \end{aligned}$$

It is useful below to have special notation \leq' for the ordering on ty_i' .

Combining the theorems about the various (err-)semilattice constructions involved in the definition of *sl* (starting from Theorem 4.12, using Lemmas 4.1 to 4.9), it is easy to prove

COROLLARY 4.13. If *P* is well-formed, then *sl* is a semilattice. Its order (written *le P m_{xs} m_{xl}*) satisfies the ascending chain condition:

$$\begin{aligned} wf\text{-}prog\ _ P &\Longrightarrow\ semilat\ (sl\ P\ mxs\ mxl) \\ wf\text{-}prog\ _ P &\Longrightarrow\ acc\ (le\ P\ mxs\ mxl) \end{aligned}$$

4.7 Applicability and Effect Instantiated

In this section, we instantiate *app* and *eff* from §4.4 for the instruction set of the Jinja VM. As for the source language we have divided the definitions into one part for normal and one part for exceptional execution.

Since the BV verifies one method at a time, we can see the context of a method and a program as fixed for the definition. The context consists of the following values:

<i>P</i>	:: <i>jvm-prog</i>	the program,
<i>C'</i>	:: <i>cname</i>	the class the method we are verifying is declared in,
<i>m_{xs}</i>	:: <i>nat</i>	maximum stack size of the method,
<i>m_{xl}</i>	:: <i>nat</i>	size of the register set for local variables,
<i>T_s</i>	:: <i>ty list</i>	types of the parameters of the method,
<i>T_r</i>	:: <i>ty</i>	return type of the method,
<i>is</i>	:: <i>instr list</i>	instructions of the method,
<i>xt</i>	:: <i>ex-table</i>	exception handler table of the method.

The context variables are proper parameters of *eff* and *app* in the Isabelle formalisation. We treat them as global here to spare the reader endless parameter lists in each definition. Formally, we use Isabelle's *locale* mechanism to hide these parameters in the presentation. Ballarin [2003] describes locales in detail.

$$\begin{aligned}
 app_i &:: instr \Rightarrow pc \Rightarrow ty_i \Rightarrow bool \\
 app_i \text{ (Load } n) pc (ST, LT) &= (n < |LT| \wedge LT_{[n]} \neq Err \wedge |ST| < maxs) \\
 app_i \text{ (Store } n) pc (T \cdot ST, LT) &= (n < |LT|) \\
 app_i \text{ (Push } v) pc (ST, LT) &= (|ST| < maxs \wedge typeof v \neq None) \\
 app_i \text{ (New } C) pc (ST, LT) &= (is-class P C \wedge |ST| < maxs) \\
 app_i \text{ (Getfield } F C) pc (T \cdot ST, LT) &= (\exists T_f. P \vdash C \text{ sees } F:T_f \text{ in } C \wedge P \vdash T \leq Class C) \\
 app_i \text{ (Putfield } F C) pc (T_1 \cdot T_2 \cdot ST, LT) &= \\
 (\exists T_f. P \vdash C \text{ sees } F:T_f \text{ in } C \wedge P \vdash T_2 \leq Class C \wedge P \vdash T_1 \leq T_f) \\
 app_i \text{ (Checkcast } C) pc (T \cdot ST, LT) &= (is-class P C \wedge is-refT T) \\
 app_i \text{ (Invoke } M n) pc (ST, LT) &= \\
 (n < |ST| \wedge \\
 (ST_{[n]} \neq NT \longrightarrow \\
 (\exists C D Ts T m. \\
 ST_{[n]} = Class C \wedge P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } D \wedge P \vdash rev (take n ST) [\leq] Ts))) \\
 app_i \text{ Return } pc (T \cdot ST, LT) &= P \vdash T \leq T_r \\
 app_i \text{ Pop } pc (T \cdot ST, LT) &= True \\
 app_i \text{ IAdd } pc (T_1 \cdot T_2 \cdot ST, LT) &= (T_1 = T_2 \wedge T_1 = Integer) \\
 app_i \text{ (Goto } b) pc s &= (0 \leq int pc + b) \\
 app_i \text{ CmpEq } pc (T_1 \cdot T_2 \cdot ST, LT) &= (T_1 = T_2 \vee is-refT T_1 \wedge is-refT T_2) \\
 app_i \text{ (IfFalse } b) pc (Boolean \cdot ST, LT) &= (0 \leq int pc + b) \\
 app_i \text{ Throw } pc (T \cdot ST, LT) &= is-refT T \\
 app_i i pc \tau &= False
 \end{aligned}$$

Fig. 23. Applicability of instructions.

4.7.1 *Normal Execution.* We begin with the definition of applicability for normal execution. The intermediate function app_i , defined in Fig. 23, works on ty_i , app will later lift it to ty_i' . The definition is parallel to *check-instr* in §3.3, it just works on types instead of values. The definition is smaller than the one of *check-instr* because some of the conditions cannot be expressed at the type level alone. These conditions are the ones that access the heap or the frame stack (most notable in the **Getfield**, **Putfield**, and **Return** instructions). It will be the responsibility of the type safety proof to show that the BV still manages to guarantee that all checks in the defensive machine are successful.

Let's take a closer look at the **IAdd** example again:

$$app_i \text{ IAdd } pc (T_1 \cdot T_2 \cdot ST, LT) = (T_1 = T_2 \wedge T_1 = Integer)$$

This is completely parallel to the defensive machine. The pattern on the left hand side ensures that there are at least two elements on the stack, and the right hand side requires that they both are integers.

The non-exceptional effect of instructions eff_i is equally simple. First, we calculate the successor program counters in Fig. 24, after that, the effect of the instruction on the type level in Fig. 25.

The successors are easy, most instructions simply proceed to $pc + 1$. The relative jumps in **IfFalse** and **Goto** use the *nat* and *int* functions to convert the HOL-

$$\begin{array}{l}
succs :: instr \Rightarrow ty_i \Rightarrow pc \Rightarrow pc \text{ list} \\
succs \text{ (IfFalse } b) \tau pc = [pc + 1, nat (int pc + b)] \\
succs \text{ (Goto } b) \tau pc = [nat (int pc + b)] \\
succs \text{ (Invoke } M n) \tau pc = (if (fst \tau)_{[n]} = NT \text{ then } [] \text{ else } [pc + 1]) \\
succs \text{ Return } \tau pc = [] \\
succs \text{ Throw } \tau pc = [] \\
succs i \tau pc = [pc + 1]
\end{array}$$

Fig. 24. Successor program counters for the non-exceptional case.

$$\begin{array}{l}
eff_i :: instr \Rightarrow ty_i \Rightarrow ty_i \\
eff_i \text{ (Load } n) (ST, LT) = (ok\text{-val } LT_{[n]} \cdot ST, LT) \\
eff_i \text{ (Store } n) (T \cdot ST, LT) = (ST, LT[n := OK T]) \\
eff_i \text{ (Push } v) (ST, LT) = (the (typeof v) \cdot ST, LT) \\
eff_i \text{ (New } C) (ST, LT) = (Class C \cdot ST, LT) \\
eff_i \text{ (Getfield } F C) (T \cdot ST, LT) = (snd (field P C F) \cdot ST, LT) \\
eff_i \text{ (Putfield } F C) (T_1 \cdot T_2 \cdot ST, LT) = (ST, LT) \\
eff_i \text{ (Checkcast } C) (T \cdot ST, LT) = (Class C \cdot ST, LT) \\
eff_i \text{ Pop } (T \cdot ST, LT) = (ST, LT) \\
eff_i \text{ IAdd } (T_1 \cdot T_2 \cdot ST, LT) = (Integer \cdot ST, LT) \\
eff_i \text{ (Goto } n) s = s \\
eff_i \text{ CmpEq } (T_1 \cdot T_2 \cdot ST, LT) = (Boolean \cdot ST, LT) \\
eff_i \text{ (IfFalse } b) (T_1 \cdot ST, LT) = (ST, LT) \\
eff_i \text{ (Invoke } M n) (ST, LT) = let Class C = ST_{[n]}; \\
\quad (_, _, T_r, _) = method P C M \\
\quad in (T_r \cdot drop (n + 1) ST, LT)
\end{array}$$

Fig. 25. Effect of instructions on the state type.

types *nat* to *int* and vice versa. **Return** and **Throw** have no successors in the same method (for the non-exceptional case). The **Invoke** *n M* instruction has no normal successor if the the stack at position *n* contains *NT*—it will always throw a *NullPointerException* exception in this case. It is different from **Getfield** and **Putfield**, because they have enough information in the instruction itself to determine the effect. **Invoke** on the other hand must rely on computed information at $ST_{[n]}$ to determine the return type of the method. In the Java VM, the **Invoke** instruction contains the static class of the method (and is thus easier to handle).

The effect eff_i on ty_i is shown in Fig. 25. The destructor *ok-val* is defined by $ok\text{-val } (OK x) = x$.

The **IAdd** instruction is in this case:

$$eff_i \text{ IAdd } (T_1 \cdot T_2 \cdot ST, LT) = (Integer \cdot ST, LT)$$

Again, as befits an abstract interpretation, the definition is completely parallel to the operational semantics, this time to *exec-instr* of the aggressive machine.

The next step is combining eff_i and *succs*:

$$\begin{array}{l}
norm\text{-eff} :: instr \Rightarrow pc \Rightarrow ty_i \Rightarrow (pc \times ty_i') \text{ list} \\
norm\text{-eff } i pc \tau \equiv map (\lambda pc'. (pc', [eff_i i \tau])) (succs i \tau pc)
\end{array}$$

$$\begin{aligned}
& \text{is-relevant-class} :: \text{instr} \Rightarrow 'm \text{ prog} \Rightarrow \text{cname} \Rightarrow \text{bool} \\
& \text{is-relevant-class} (\text{Getfield } F D) P C = P \vdash \text{NullPointer} \preceq^* C \\
& \text{is-relevant-class} (\text{Putfield } F D) P C = P \vdash \text{NullPointer} \preceq^* C \\
& \text{is-relevant-class} (\text{Checkcast } D) P C = P \vdash \text{ClassCast} \preceq^* C \\
& \text{is-relevant-class} (\text{New } D) P C = P \vdash \text{OutOfMemory} \preceq^* C \\
& \text{is-relevant-class} \text{Throw } P C = \text{True} \\
& \text{is-relevant-class} (\text{Invoke } M n) P C = \text{True} \\
& \text{is-relevant-class } i P C = \text{False} \\
\\
& \text{is-relevant-entry} :: 'm \text{ prog} \Rightarrow \text{instr} \Rightarrow \text{pc} \Rightarrow \text{ex-entry} \Rightarrow \text{bool} \\
& \text{is-relevant-entry } P i \text{ pc } e \equiv \text{let } (f, t, C, h, d) = e \text{ in } \text{is-relevant-class } i P C \wedge \text{pc} \in \{f..<t\} \\
\\
& \text{relevant-entries} :: 'm \text{ prog} \Rightarrow \text{instr} \Rightarrow \text{pc} \Rightarrow \text{ex-table} \Rightarrow \text{ex-table} \\
& \text{relevant-entries } P i \text{ pc} \equiv \text{filter } (\text{is-relevant-entry } P i \text{ pc})
\end{aligned}$$

Fig. 26. Finding relevant exception handlers.

The result is a list of edges in the control flow graph determined by succs , each of them marked with the result of eff_i . As we use norm-eff only for reachable instructions below, we can safely mark the successors as reachable with $[-]$.

We now have two functions app_i and norm-eff describing normal execution in the bytecode verifier. The next section turns to exceptions.

4.7.2 Exceptions. Abstractly, exceptions merely add more edges to the control flow graph. In the JVM (Jinja as well as Java), these edges must all lead to the start of the exception handler that is relevant for the current instruction. Only at runtime this relevant handler is uniquely determined, statically we must consider a number of handlers, because relevance of a handler depends on which exception was raised. The `Invoke` instruction for instance may raise a `NullPointerException` exception, or it may propagate an exception up that was thrown in the invoked method. For each of these cases a different handler might be relevant. As for the conditional branch instruction, the bytecode verifier simply checks all of them.

Thus, the first thing we need to do is determine which handlers might be relevant for an instruction. Figure 26 shows this in three stages: $\text{relevant-class } i P C$ is `True` iff i can raise an exceptions of class C , $\text{is-relevant-entry } P i \text{ pc } e$ is `True` iff entry e in the exception table might match instruction i at position pc , and finally $\text{relevant-entries } P i \text{ pc } xt$ is the list of exception table entries that are relevant for instruction i at position pc .

For applicability in the exception case we require that the class name mentioned in the exception handler is indeed a declared class, and that the stack is between mxs and the number d of entries the exception handler expects.

$$\begin{aligned}
& \text{xcpt-app} :: \text{instr} \Rightarrow \text{pc} \Rightarrow \text{ty}_i \Rightarrow \text{bool} \\
& \text{xcpt-app } i \text{ pc} (ST, LT) \equiv \\
& \forall (f, t, C, h, d) \in \text{set } (\text{relevant-entries } P i \text{ pc } xt). \text{is-class } P C \wedge d \leq |ST| \wedge d < \text{mxs}
\end{aligned}$$

The effect of instructions in the exception case is equally simple. Each edge leads to the start of the exception handler, and the local variables are unchanged. We cut down the operand stack to d elements of the current stack, and push the exception object on top.

$$\begin{aligned}
& \text{xcpt-eff} :: \text{instr} \Rightarrow \text{pc} \Rightarrow \text{ty}_i \Rightarrow (\text{pc} \times \text{ty}_i') \text{ list} \\
& \text{xcpt-eff } i \text{ pc} (ST, LT) \equiv
\end{aligned}$$

$map (\lambda(f, t, C, h, d). (h, [(Class\ C \cdot drop\ (|ST| - d)\ ST, LT])))\ (relevant\ entries\ P\ i\ pc\ xt)$

4.7.3 Combining Normal and Exceptional Execution. Combining the normal and exceptional case, we can now build the full effect function: if an instruction is unreachable, it has no outgoing edges; if it is reachable the overall effect is simply an append of the normal and the exception case.

$eff :: instr \Rightarrow pc \Rightarrow ty_i' \Rightarrow (pc \times ty_i')\ list$
 $eff\ i\ pc\ t \equiv case\ t\ of\ None \Rightarrow []\ |\ [\tau] \Rightarrow norm\text{-}eff\ i\ pc\ \tau\ @\ xcpt\text{-}eff\ i\ pc\ \tau$

For applicability we have: an instruction is applicable if it is unreachable (then it can do no harm) or if it is applicable in the normal and in the exception case. Additionally, we require that the pc does not leave the instruction sequence.

$app :: instr \Rightarrow pc \Rightarrow ty_i' \Rightarrow bool$
 $app\ i\ pc\ t \equiv$
 $case\ t\ of\ None \Rightarrow True$
 $|\ [\tau] \Rightarrow app_i\ i\ pc\ \tau \wedge xcpt\text{-}app\ i\ pc\ \tau \wedge (\forall (pc', \tau') \in set\ (eff\ i\ pc\ t). pc' < |is|)$

4.8 Well-typings

Having defined the semilattice and the transfer function in §4.6 and §4.7, we show in this section how the parts are put together to get a definition of well-typings for the Jinja VM.

The abstract framework gives us a predicate *wt-app-eff* (§4.4) describing well-typings $\tau s :: ty_i'\ list$ as method types that fit an instruction sequence. To obtain type-safety and an executable bytecode verifier, we additionally require a start condition for instruction 0 (at method invocation) and some side conditions explained below.

The operational semantics of **Invoke** tells us the start condition at method invocation: the stack is empty, the first register contains the *this* pointer (of type *Class C'*), the next registers contain the parameters of the method, and the rest of the registers are reserved for local variables (which do not have a value yet). Note that the definitions are still in the context of a fixed method as defined in §4.7, so C' is the class to be verified, Ts are the parameters, and mxl_0 the number of local variables (which is related to mxl of §4.7 by $mxl = 1 + |Ts| + mxl_0$). The \leq' is the semilattice order on ty_i' of §4.6.

$wt\text{-}start\ \tau s \equiv P \vdash [([], OK\ (Class\ C') \cdot map\ OK\ Ts\ @\ replicate\ mxl_0\ Err)] \leq' \tau s_{[0]}$

The method type τs is a well-typing for a method if it satisfies *wt-method*. To define it, we instantiate *wt-app-eff* from the framework with $\lambda pc\ \tau. app\ is_{[pc]}\ pc\ \tau$ for the abstract *app*, and $\lambda pc\ \tau. eff\ is_{[pc]}\ pc\ \tau$ for the abstract *eff*.

$wt\text{-}method\ \tau s =$
 $(is \neq [] \wedge |\tau s| = mpc \wedge OK\ 'set\ \tau s \subseteq states\ P\ mxs\ mxl \wedge wt\text{-}start\ \tau s \wedge wt\text{-}app\text{-}eff\ \tau s)$

The *states* are the carrier set of the semilattice. Remember that the method type τs does not contain the *Err* layer of the semilattice, hence we take the image $OK\ 'set\ \tau s$ of *set* τs under *OK*. For well-typedness *wt-method* also requires that the method contains at least one instruction, and that the method type covers all instructions.

It is occasionally useful in the proofs and also for the compiler below, to define the notion of an instruction being well-typed. This is just the matrix of the $\forall p < |\tau s|$ in *wt-app-eff* (c.f. §4.4). Since it will be used outside the fixed method context of this section, it has more parameters.

$$P, T_r, mxs, mpc, xt \vdash i, pc :: \tau s \equiv \\ app \ i \ pc \ \tau s_{[pc]} \wedge (\forall (pc', \tau') \in set \ (eff \ i \ pc \ \tau s_{[pc]}). P \vdash \tau' \leq' \tau s_{[pc]})$$

The notation $P, T_r, mxs, mpc, xt \vdash i, pc :: \tau s$ is read as: in the context of program P , return type T_r , maximum stack size mxs , number of instructions mpc and exception table xt , the instruction i at position pc is well-typed w.r.t. method type $\tau s :: ty_i'$ list.

It remains to lift well-typings from methods to programs. Well-typings of programs are functions $\Phi :: ty_P$ with

$$\mathbf{types} \ ty_P = cname \Rightarrow mname \Rightarrow ty_i' \ list$$

These functions return a well-typing for each class and method in the program. A Jinja VM program is well-formed according to Φ if each method body is well-typed. At this point, the full parameter list of *wt-method* becomes visible.

$$wf\text{-jvm}\text{-prog}_\Phi \equiv \\ wf\text{-prog} \ (\lambda P \ C \ (M, Ts, T_r, mxs, mxl_0, is, xt). wt\text{-method} \ P \ C \ Ts \ T_r \ mxs \ mxl_0 \ is \ xt \ (\Phi \ C \ M)) \\ wf\text{-jvm}\text{-prog} \ P \equiv \exists \Phi. wf\text{-jvm}\text{-prog}_\Phi \ P$$

4.9 An Executable Bytecode Verifier

In §4.8 we defined well-typings for the Jinja VM. This section shows how to instantiate the type inference algorithm of §4.5 to get an executable bytecode verifier for Jinja.

With the semilattice as defined in §4.6 and the transfer function of §4.7, and still within the same method context as for *wt-method*, we only need to provide the correct start value to Kildall's algorithm to get an executable BV:

$$wt\text{-kildall} \equiv \\ is \neq [] \wedge \\ (\text{let } \tau_0 = [([], [OK \ (Class \ C')]) \ @ \ map \ OK \ Ts \ @ \ replicate \ mxl_0 \ Err]); \\ \tau s_0 = OK \ \tau_0 \cdot replicate \ (|is| - 1) \ (OK \ None) \\ in \ \forall n < |is|. (kildall \ \tau s_0)_{[n]} \neq Err)$$

Position 0 in τs_0 is the same as the start value in *wt-start*. Since we know nothing yet about the positions greater than 0, we fill in the bottom element *OK None* for those.

Lifting to full programs and filling in the method context is the same as for *wt-method*:

$$wf\text{-jvm}\text{-prog}_k \ P \equiv \\ wf\text{-prog} \ (\lambda P \ C' \ (M, Ts, T_r, mxs, mxl_0, is, xt). wt\text{-kildall} \ P \ C' \ Ts \ T_r \ mxs \ mxl_0 \ is \ xt) \ P$$

This definition only gives us a working BV if *step* meets the conditions of Theorem 4.11. We have shown that the transfer function *step*, built from *app* and *eff* as described in §4.4 and §4.7, is monotone, bounded, and type preserving (w.r.t. *states* and $|is|$). Albeit large (a case distinction over the instruction set), the proof that *step* is monotone and type preserving is easy and mostly automatic. That *step* is bounded is checked explicitly by the *app* component of *step*.

Using Theorem 4.11, we have then proved the following:

THEOREM 4.14. The executable BV is sound and recognises all well-typed programs:

$$wf\text{-}jvm\text{-}prog_k P = wf\text{-}jvm\text{-}prog P$$

4.10 Type Safety

This section is about the type safety of the well-typing specification above. The type safety theorem states that the bytecode verifier is correct, that it guarantees safe execution. If the bytecode verifier succeeds and we start the program P in its canonical start state (§3.3), the defensive Jinja VM will never return a type error. With Theorem 3.1, this implies that the checks of the defensive machine are redundant and the aggressive machine can be used safely instead.

THEOREM 4.15. If C is a class in P with a method M , formally $P \vdash C \text{ sees } M: [] \rightarrow T = b \text{ in } C$, and if $wf\text{-}jvm\text{-}prog P$, then

$$P \vdash Normal (start\text{-}state P C M) \xrightarrow{djvm} \sigma' \implies \sigma' \neq TypeError$$

To prove this theorem, we set out from a program P for which the bytecode verifier returns true, i.e., for which there is a Φ such that $wf\text{-}jvm\text{-}prog_\Phi P$ holds. The proof builds on the observation that all runtime states σ that conform to the types in Φ are type safe. For σ **conforms to** Φ , we write $P, \Phi \vdash \sigma \checkmark$. For $P, \Phi \vdash \sigma \checkmark$ to be true, the following must hold: if in state σ execution is at position pc of method M in C , then the state type $(\Phi C M)_{[pc]}$ must be of the form $[\tau]$, and for every value v on the stack or in the register set the type of v must be a subtype of the corresponding entry in its static counterpart τ . This is the essence of the conformance relation. To show that it is invariant during execution, it needs to be strengthened. We show the complete formal definition of conformance below, but to avoid getting bogged down in detail we continue to sketch the general proof outline first. For this strong conformance we have shown that it is invariant during execution if the program is well-typed.

LEMMA 4.16. If $wf\text{-}jvm\text{-}prog_\Phi P$ and $P \vdash \sigma \xrightarrow{jvm} \sigma'$ and $P, \Phi \vdash \sigma \checkmark$ then $P, \Phi \vdash \sigma' \checkmark$.

The proof is by induction on the length of the execution and then by case distinction on the instruction set. For each instruction, we conclude from the conformance of σ together with the *app* part of $wf\text{-}jvm\text{-}prog$ that all assumptions of the operational semantics are met (like “the stack is not empty”). Then we execute the instruction and observe that the new state σ' conforms to the corresponding τ' in *eff* pc τ . This invariance lemma corresponds to *subject reduction* in a traditional small step semantics.

Lemma 4.16 is still not sufficient for type safety, though: it might be the case that *start* $P C M$ does not conform to Φ . We have shown that this is not so.

LEMMA 4.17. If $wf\text{-}jvm\text{-}prog_\Phi P$ and $P \vdash C \text{ sees } M: [] \rightarrow T = m \text{ in } C$ then $P, \Phi \vdash start\text{-}state P C M \checkmark$.

Lemmas 4.16 and 4.17 together say that all states that occur in any execution of program P conform to Φ if we start P in the canonical way.

The last step in the proof of Theorem 4.15 is Lemma 4.18. It corresponds to *progress* in a traditional small step semantics.

LEMMA 4.18. An execution step started in a conformant state cannot produce a type error in well-typed programs:

If $wf\text{-}jvm\text{-}prog_{\Phi} P$ and $P, \Phi \vdash \sigma \surd$ then $exec_d P \sigma \neq \text{TypeError}$.

The proof of Lemma 4.18 is by case distinction on the current instruction in σ . Similar to the proof of Lemma 4.16, the conformance relation together with the *app* part of *wf-jvm-prog* ensures *check-instr* in $exec_d$ returns true. Because we know that all states during execution conform, we can conclude Theorem 4.15: there will be no type errors in well-typed programs.

We now show the formal definition of the conformance relation between dynamic Jinja VM states and static states (types) in the Jinja BV.

For the proof of the invariance lemma (Lemma 4.16) to go through, the intuitive notion of conformance we have given above is not sufficient, the formal conformance relation $P, \Phi \vdash \sigma \surd$ is stronger. It describes in detail the states that can occur during execution, the form of the heap, and the form of the method invocation stack.

We begin by lifting the single value conformance $P, h \vdash v : \leq T$ of §2.9 to stack and local variables. For the stack this is just the pointwise lifting of $: \leq$ to lists. As usual, we write $P, h \vdash vs [: \leq] ST$ for this. For the local variables we first need to treat the *Err* level:

$$P, h \vdash v : \leq_{\top} E \equiv \text{case } E \text{ of } Err \Rightarrow \text{True} \mid OK \ T \Rightarrow P, h \vdash v : \leq T$$

Lifting to lists $P, h \vdash vs [: \leq_{\top}] LT$ is then canonical again.

A call frame conforms if its stack and register set conform, and if the program counter lies inside the instruction list.

$$\begin{aligned} \text{conf-f } P \ h \ (ST, LT) \text{ is } (stk, loc, C, M, pc) &\equiv \\ P, h \vdash stk [: \leq] ST \wedge P, h \vdash loc [: \leq_{\top}] LT \wedge pc < |is| \end{aligned}$$

This is still not enough. For proving the **Return** case, we also need information about the structure of the call frame stack. The predicate *conf-fs* below describes the structure of the call frame stack beneath the topmost frame. The parameters M_0 , n_0 and T_0 are name, number of parameters and return type of the method in the topmost frame.

$$\begin{aligned} \text{conf-fs } P \ h \ \Phi \ M_0 \ n_0 \ T_0 \ [] &= \text{True} \\ \text{conf-fs } P \ h \ \Phi \ M_0 \ n_0 \ T_0 \ ((stk, loc, C, M, pc) \cdot fs) &= \\ (\exists ST \ LT \ Ts \ T \ mxs \ mxl_0 \ is \ xt. & \\ (\Phi \ C \ M)_{[pc]} = [(ST, LT)] \wedge P \vdash C \text{ sees } M : Ts \rightarrow T = (mxs, mxl_0, is, xt) \text{ in } C \wedge & \\ (\exists D \ Ts' \ T' \ m \ D'. & \\ is_{[pc]} = \text{Invoke } M_0 \ n_0 \wedge ST_{[n_0]} = \text{Class } D \wedge P \vdash D \text{ sees } M_0 : Ts' \rightarrow T' = m \text{ in } D' \wedge & \\ P \vdash T_0 \leq T') \wedge & \\ \text{conf-f } P \ h \ (ST, LT) \text{ is } (stk, loc, C, M, pc) \wedge & \\ \text{conf-fs } P \ h \ \Phi \ M \ |Ts| \ T \ fs) & \end{aligned}$$

In the definition above, a list of call frames conforms if it is empty. If it is not empty, then for the head frame the following must hold: the state type $(\Phi \ C \ M)_{[pc]}$ for the current instruction must denote a reachable instruction; the call frame must belong to a defined method; it must be halted at the **Invoke** instruction which created the

call frame above (this is not easily expressed without access to something like a call history, but it is enough to demand that the M and n in `Invoke M n` are M_0 and n_0 , and that the return type of a lookup on the class D in $ST_{[n_0]}$ conforms to T_0 , the return type the frame above expects); finally, the current frame and the rest of the call frame stack must conform.

The following is the top level conformance relation between a state and a program type. The first two cases are trivial, the third case requires a conformant heap ($P \vdash h \checkmark$), contains special handling for the topmost call frame and delegates the rest to *conf-fs*. The topmost frame is special because it does not need to be halted at an `Invoke` instruction. The topmost frame must conform and the current state type must denote a reachable instruction. The method lookup provides *conf-f* and *conf-fs* with the required parameters.

$$\begin{aligned}
P, \Phi \vdash (\text{None}, h, []) \checkmark &= \text{True} \\
P, \Phi \vdash ([x], h, fs) \checkmark &= (fs = []) \\
P, \Phi \vdash (\text{None}, h, (stk, loc, C, M, pc) \cdot fs) \checkmark &= \\
(\exists Ts \ T \ mxs \ mxl_0 \ is \ xt \ s. & \\
P \vdash h \checkmark \wedge P \vdash C \ \text{sees } M: Ts \rightarrow T = (mxs, mxl_0, is, xt) \ \text{in } C \wedge & (\Phi \ C \ M)_{[pc]} = [s] \wedge \\
\text{conf-f } P \ h \ s \ is \ (stk, loc, C, M, pc) \wedge & \\
\text{conf-fs } P \ h \ \Phi \ M \ |Ts| \ T \ fs) &
\end{aligned}$$

Fig. 27 is a snapshot of the Jinja VM state in the middle of a typical program execution. On the left there is the Jinja VM with its frame stack and heap, on the right there are the method types the BV predicted for this program. The program declarations appear on the lower right side in the static part.

The state on the left in Fig. 27 conforms to the static type information on the right: all objects in the heap conform, because the values of the field F (declared in class B) are all of type *Class A* (*Null* is of type *Class A*, and the address *Addr 0* points to an object of *Class B* which is a subclass of *Class A*). All frames but the topmost one are halted at the `Invoke` instruction that created the next frame. The dynamic operand stacks conform to the static ones, because their length is the same and all values have conforming type. The topmost frame conforms, too, because its *pc* points to a valid instruction (`Getfield F B`), and the value on the dynamic operand stack is an address that points to an object of class C .

4.11 Related Work

The bytecode verifier presented here is a simplified version of the one by Klein [2003] which in turn builds on the work by Nipkow [2001] and Pusch [1999]. Most closely related to these is the work by Barthe and Dufay [2004] who use the Coq system for a formalisation of the JavaCard virtual machine and its BV. They formalise the full JavaCard bytecode language, but with a simplified treatment of some features like bytecode subroutines.

The ASM formalisation of Stärk et al. [2001] contains almost all of the Java BV's features and is executable. It does not use mechanical support or checking for its proofs.

Paper formalisations of Java bytecode verification begin with Stata and Abadi [1998] who give a first formal type system for the JVM. Freund and Mitchell [2003] and Freund [2000] build on this type system and extend it to a substantial subset of the JVM.

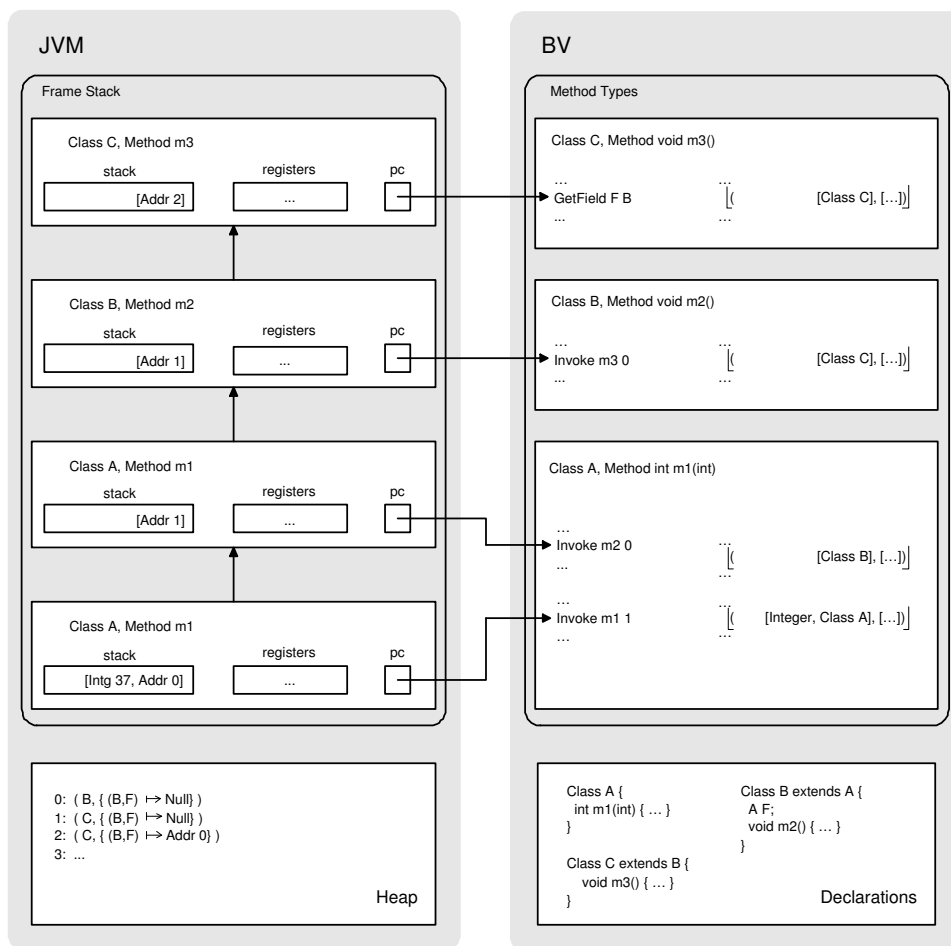


Fig. 27. Jinja VM execution snapshot.

Working towards a verified implementation in SPECWARE, Coglio et al. [2000] have specified and analysed large portions of the bytecode verifier. Goldberg [1998] rephrases and generalises the overly concrete description of the BV given in the JVM specification [Lindholm and Yellin 1999] as an instance of a generic data flow framework. Qian [2000] also proves the correctness of an algorithm for turning his type checking rules into a data flow analyser. However, his algorithm is still quite abstract. The formulation of the bytecode verification algorithm as a direct instance of an abstract data flow framework first appears in the work by Nipkow [2001] and is refined by Klein and Nipkow [2003] and Klein [2003].

There is an interesting variant of type inference for the JVM, namely lightweight bytecode verification [Rose and Rose 1998; Rose 2002; 2003]. It uses a certificate to reduce time and space usage during type inference. For space reasons we have left it out here, but lightweight bytecode verification fits in nicely with our formalisation of the VM type system [Klein and Nipkow 2001]. Moreover, it can be formulated as an

inference algorithm in the same abstract framework that we use for Kildall’s data flow analyser [Klein 2003]. Other variations of lightweight bytecode verification algorithms are surveyed by Leroy [2003].

There is a large body of literature on bytecode verification that concentrates on special language features like subroutines [Coglio 2004; Klein and Wildmoser 2003], object initialisation, dynamic class loading, thread monitors and access control. There are also a number of more or less radically different approaches to type inferences like model checking, reducing the BV to Haskell type inference, and using ML-style polymorphism. We refer to the overview articles for these.

5. COMPILER

Our compiler operates in two stages: first it replaces variable names in expressions by indices (§5.2), then it generates the actual JVM code (§5.3). Once we have lifted compilation of expressions to the level of programs (§5.4) we show that both compiler stages preserve the big-step semantics (§5.5 and §5.6). Finally we show that not only the semantics, but also the well-typedness of expressions and well-formedness of programs is preserved by compilation, i.e. that the compiler produces well-typed JVM code (§5.8 and §5.9).

5.1 Intermediate Language

The intermediate language of expressions is called $expr_1$ and is identical to $expr$, except that all variable names are replaced by natural numbers, except in field access and field assignment. To avoid duplication, in Isabelle/HOL we have defined one parameterised data type $'a\ exp$, and $expr$ and $expr_1$ are abbreviations for the instances $vname\ exp$ and $nat\ exp$. The type of J_1 programs is defined by

$$\mathbf{types}\ J_1\text{-prog} = expr_1\ prog$$

In contrast to $J\text{-prog}$, methods no longer need parameter names because they have been replaced by numbers.

5.1.1 Big step semantics. The motivation for our choice of intermediate language is that its semantics is based on almost the same state space as the JVM: local variables are no longer stored in a map but in a fixed length list of values, i.e. in an array. Its big step semantics is of the form $P \vdash_1 \langle e, s \rangle \Rightarrow \langle e', s' \rangle$ where $P :: J_1\text{-prog}$, $e, e' :: expr_1$ and $s, s' :: heap \times val\ list$.

Most of the evaluation rules for $expr_1$ are the same as the ones for $expr$. The ones that differ are shown in Fig. 28. The modifications in the rules for variable access, variable assignment and exception catching simply reflect the difference between a map and an array. Note that the rules are defensive in that they require the index to be within the bounds of the array. The rule for blocks expresses that all local variables are pre-allocated and blocks are irrelevant for the semantics. However, they are not irrelevant for the type system because of the types of local variables.

The method call rule has changed because the state in which the body is evaluated needs to be initialized with an array (ls_2'). The initial section of that array contains the address of the object and the parameters, the remainder is initialized with an arbitrary value, just as in the semantics of `Invoke` in the JVM (see §3.2). To find

$$\begin{array}{c}
 \frac{ls_{[i]} = v \quad i < |ls|}{P \vdash_1 \langle \mathbf{Var} \ i, (h, ls) \rangle \Rightarrow \langle \mathbf{Val} \ v, (h, ls) \rangle} \\
 \\
 \frac{P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \mathbf{Val} \ v, (h, ls) \rangle \quad i < |ls| \quad ls' = ls[i := v]}{P \vdash_1 \langle i := e, s_0 \rangle \Rightarrow \langle \mathbf{unit}, (h, ls') \rangle} \\
 \\
 \frac{P \vdash_1 \langle e_1, s_0 \rangle \Rightarrow \langle \mathbf{Throw} \ a, (h_1, ls_1) \rangle \quad h_1 \ a = \lfloor (D, fs) \rfloor}{P \vdash D \preceq^* C \quad i < |ls_1| \quad P \vdash_1 \langle e_2, (h_1, ls_1[i := \mathbf{Addr} \ a]) \rangle \Rightarrow \langle e_2', (h_2, ls_2) \rangle} \\
 \frac{P \vdash_1 \langle \mathbf{try} \ e_1 \ \mathbf{catch} \ (C \ i) \ e_2, s_0 \rangle \Rightarrow \langle e_2', (h_2, ls_2) \rangle}{} \\
 \\
 \frac{P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle e', s_1 \rangle}{P \vdash_1 \langle \{i:T; e\}, s_0 \rangle \Rightarrow \langle e', s_1 \rangle} \\
 \\
 \frac{P \vdash_1 \langle e, s_0 \rangle \Rightarrow \langle \mathbf{addr} \ a, s_1 \rangle \quad P \vdash_1 \langle es, s_1 \rangle [\Rightarrow] \langle \mathbf{map} \ \mathbf{Val} \ vs, (h_2, ls_2) \rangle}{\begin{array}{l} h_2 \ a = \lfloor (C, fs) \rfloor \quad P \vdash C \ \mathbf{sees} \ M: Ts \rightarrow T = \mathbf{body} \ \mathbf{in} \ D \\ |vs| = |Ts| \quad ls_2' = \mathbf{Addr} \ a \cdot vs \ @ \ \mathbf{replicate} \ (\mathbf{max}\text{-}\mathbf{vars} \ \mathbf{body}) \ \mathbf{arbitrary} \\ P \vdash_1 \langle \mathbf{body}, (h_2, ls_2') \rangle \Rightarrow \langle e', (h_3, ls_3) \rangle \end{array}} \\
 \frac{P \vdash_1 \langle e.M(es), s_0 \rangle \Rightarrow \langle e', (h_3, ls_2) \rangle}{}
 \end{array}$$

 Fig. 28. Evaluation rules for $expr_1$ differing from those for $expr$

$$\begin{array}{l}
 \mathcal{B} \ (\mathbf{new} \ C) \ i = \mathbf{True} \\
 \mathcal{B} \ (\mathbf{Cast} \ C \ e) \ i = \mathcal{B} \ e \ i \\
 \mathcal{B} \ (\mathbf{Val} \ v) \ i = \mathbf{True} \\
 \mathcal{B} \ (e_1 \ll \mathbf{bop} \gg e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \\
 \mathcal{B} \ (\mathbf{Var} \ j) \ i = \mathbf{True} \\
 \mathcal{B} \ (e.F\{D\}) \ i = \mathcal{B} \ e \ i \\
 \mathcal{B} \ (j := e) \ i = \mathcal{B} \ e \ i \\
 \mathcal{B} \ (e_1.F\{D\} := e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \\
 \mathcal{B} \ (e.M(es)) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} \ s \ es \ i) \\
 \mathcal{B} \ \{j:T; e\} \ i = (i = j \wedge \mathcal{B} \ e \ (i + 1)) \\
 \mathcal{B} \ (e_1; e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \\
 \mathcal{B} \ (\mathbf{if} \ (e) \ e_1 \ \mathbf{else} \ e_2) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} \ e_1 \ i \wedge \mathcal{B} \ e_2 \ i) \\
 \mathcal{B} \ (\mathbf{throw} \ e) \ i = \mathcal{B} \ e \ i \\
 \mathcal{B} \ (\mathbf{while} \ (e) \ c) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} \ c \ i) \\
 \mathcal{B} \ (\mathbf{try} \ e_1 \ \mathbf{catch} \ (C \ j) \ e_2) \ i = (\mathcal{B} \ e_1 \ i \wedge i = j \wedge \mathcal{B} \ e_2 \ (i + 1)) \\
 \mathcal{B} \ s \ \square \ i = \mathbf{True} \\
 \mathcal{B} \ s \ (e \cdot es) \ i = (\mathcal{B} \ e \ i \wedge \mathcal{B} \ s \ es \ i)
 \end{array}$$

 Fig. 29. Definition of \mathcal{B}

out how many local variables are needed, $\mathbf{max}\text{-}\mathbf{vars}$ (Fig. 30) computes the maximal depth of nested local variables. However, the depth may say nothing about the actual indices, for example in $\{29:T_0; \{17:T_1; e\}\}$. But the layout of ls_2' assumes the two are related: all indices in \mathbf{body} must be below $1 + \mathbf{size} \ \mathbf{vs} + \mathbf{max}\text{-}\mathbf{vars} \ \mathbf{body}$. This assumption is justified because, as we will see later on, stage 1 of the compiler produces intermediate expressions with the following property: starting from some given index, say 4, indices increase by 1 for each nested block. For example, $\{5:T_0; \{6:T_1; e_1\}; \{6:T_2; e_2\}\}$ is fine, but $\{6:T_1; \{5:T_0; e\}\}$, $\{5:T_0; \{7:T_2; e\}\}$ and $\{6:T_1; \{7:T_0; e\}\}$ are not. This property is formalized by \mathcal{B} in Fig. 29 and could be called an inverse de Bruijn numbering scheme.

$max\text{-vars } (\mathbf{new } C) = 0$
 $max\text{-vars } (\mathbf{Cast } C e) = max\text{-vars } e$
 $max\text{-vars } (\mathbf{Val } v) = 0$
 $max\text{-vars } (e_1 \ll bop \gg e_2) = max (max\text{-vars } e_1) (max\text{-vars } e_2)$
 $max\text{-vars } (\mathbf{Var } V) = 0$
 $max\text{-vars } (V := e) = max\text{-vars } e$
 $max\text{-vars } (e.F\{D\}) = max\text{-vars } e$
 $max\text{-vars } (e_1.F\{D\} := e_2) = max (max\text{-vars } e_1) (max\text{-vars } e_2)$
 $max\text{-vars } (e.M(es)) = max (max\text{-vars } e) (max\text{-varss } es)$
 $max\text{-vars } \{V:T; e\} = max\text{-vars } e + 1$
 $max\text{-vars } (e_1; e_2) = max (max\text{-vars } e_1) (max\text{-vars } e_2)$
 $max\text{-vars } (\mathbf{if } (e) e_1 \mathbf{else } e_2) = max (max\text{-vars } e) (max (max\text{-vars } e_1) (max\text{-vars } e_2))$
 $max\text{-vars } (\mathbf{while } (b) e) = max (max\text{-vars } b) (max\text{-vars } e)$
 $max\text{-vars } (\mathbf{throw } e) = max\text{-vars } e$
 $max\text{-vars } (\mathbf{try } e_1 \mathbf{catch } (C V) e_2) = max (max\text{-vars } e_1) (max\text{-vars } e_2 + 1)$
 $max\text{-varss } [] = 0$
 $max\text{-varss } (e \cdot es) = max (max\text{-vars } e) (max\text{-varss } es)$

Fig. 30. Definition of $max\text{-vars}$

$$\begin{array}{c}
\frac{E_{[i]} = T \quad i < |E|}{P, E \vdash_1 \mathbf{Var } i :: T} \\
\\
\frac{E_{[i]} = T \quad i < |E| \quad P, E \vdash_1 e :: T' \quad P \vdash T' \leq T}{P, E \vdash_1 i := e :: \mathbf{Void}} \\
\\
\frac{is\text{-type } P T \quad P, E @ [T] \vdash_1 e :: T'}{P, E \vdash_1 \{i:T; e\} :: T'} \\
\\
\frac{P, E \vdash_1 e_1 :: T \quad P, E @ [\mathbf{Class } C] \vdash_1 e_2 :: T \quad is\text{-class } P C}{P, E \vdash_1 \mathbf{try } e_1 \mathbf{catch } (C i) e_2 :: T}
\end{array}$$

Fig. 31. Typing rules for $expr_1$ differing from those for $expr$

5.1.2 *Type system.* Again we replace a map by a list: in the judgment $P, E \vdash_1 e :: T$, the environment E is now of type *ty list* (and $P :: J_1\text{-prog}$). Most rules for this type system are identical to the ones for $expr$. The ones that have changed are shown in Fig. 31. They also rely on the inverse de Bruijn numbering scheme by ignoring the i in $\{i:T; e\}$.

5.1.3 *Well-formedness.* Well-formedness of J_1 programs is essentially well-formedness of Jinja programs (§2.8) where the conditions on the parameter names have been dropped and inverse de Bruijn numbering is required:

$$\begin{array}{l}
wf\text{-}J_1\text{-mdecl } P C (M, Ts, T, body) \equiv \\
(\exists T'. P, \mathbf{Class } C \cdot Ts \vdash_1 body :: T' \wedge P \vdash T' \leq T) \wedge \mathcal{D} \text{ body } [\{..|Ts|\}] \wedge \mathcal{B} \text{ body } (|Ts| + 1)
\end{array}$$

$$wf\text{-}J_1\text{-prog} \equiv wf\text{-prog } wf\text{-}J_1\text{-mdecl}$$

In the definition of \mathcal{D} (§2.7) we pretended that it operates on $expr$, but in reality it is defined on the polymorphic type *'a exp* and hence works on $expr_1$ as well.

```

compE1 Vs (new C) = new C
compE1 Vs (Cast C e) = Cast C (compE1 Vs e)
compE1 Vs (Val v) = Val v
compE1 Vs (e1 <<bop>> e2) = compE1 Vs e1 <<bop>> compE1 Vs e2
compE1 Vs (Var V) = Var (index Vs V)
compE1 Vs (V := e) = index Vs V := compE1 Vs e
compE1 Vs (e.F{D}) = compE1 Vs e.F{D}
compE1 Vs (e1.F{D} := e2) = compE1 Vs e1.F{D} := compE1 Vs e2
compE1 Vs (e.M(es)) = compE1 Vs e.M(compEs1 Vs es)
compE1 Vs {V:T; e} = {|Vs|:T; compE1 (Vs @ [V]) e}
compE1 Vs (e1; e2) = compE1 Vs e1; compE1 Vs e2
compE1 Vs (if (e) e1 else e2) = if (compE1 Vs e) compE1 Vs e1 else compE1 Vs e2
compE1 Vs (while (e) c) = while (compE1 Vs e) compE1 Vs c
compE1 Vs (throw e) = throw (compE1 Vs e)
compE1 Vs (try e1 catch (C V) e2) =
try compE1 Vs e1 catch (C |Vs|) compE1 (Vs @ [V]) e2
compEs1 Vs [] = []
compEs1 Vs (e · es) = compE1 Vs e · compEs1 Vs es

```

Fig. 32. Definition of *compE*₁

5.2 Stage 1: Names to Indices

The translation from *expr* to *expr*₁ is parametrised by the list of variables declared on the path from the root of the expression to the current subexpression. The **index** of a variable is the position of its rightmost occurrence in that list: in [A, B, A, C, D], A has index 2, B index 1, and C index 3. The formal definition

```

index [] y = 0
index (x · xs) y = (if x = y then if x ∈ set xs then index xs y + 1 else 0 else index xs y + 1)

```

reveals that the index of an element not in the list is the length of the list. Function *compE*₁ :: *vname list* ⇒ *expr* ⇒ *expr*₁ (Fig. 32) traverses an expression and translates variable names via *index*.

5.3 Stage 2: Code Generation

Code generation comprises the generation of an instruction list and of an exception table. Generating the instruction list (function *compE*₂ :: *expr*₁ ⇒ *instr list* in Fig. 33) is straightforward since we are already on the level of indices rather than names. The only case that may need some explanations is try-catch. We compile the code as suggested in the official JVM specification [Lindholm and Yellin 1999]: first the try block, then a jump over the rest to the end, then (and this is the entry point for the exception handler) the exception reference is stored in the local variable, and finally the catch block. Remember that *int* is the injection from naturals (produced by the length function) into integers (required by branch instructions).

Producing the code is simplified because branch instructions are relative. Hence the compiler does not need to compute absolute address. The exception table, however, contains absolute addresses. Hence its generation (function *compxE*₂ :: *expr*₁ ⇒ *pc* ⇒ *nat* ⇒ *ex-table* in Fig. 34) requires the current program counter as a parameter. The final parameter is the current size of the stack. Function *compxE*₂ traverses the expression top-down and left-to-right, collecting the handler information from all try-catch constructs it encounters, while keeping track of the

```

compE2 (new C) = [New C]
compE2 (Cast C e) = compE2 e @ [Checkcast C]
compE2 (Val v) = [Push v]
compE2 (e1 <<bop>> e2) = compE2 e1 @ compE2 e2 @ (case bop of = => [CmpEq] | + => [IAdd])
compE2 (Var i) = [Load i]
compE2 (i := e) = compE2 e @ [Store i, Push Unit]
compE2 (e.F{D}) = compE2 e @ [Getfield F D]
compE2 (e1.F{D} := e2) = compE2 e1 @ compE2 e2 @ [Putfield F D, Push Unit]
compE2 (e.M(es)) = compE2 e @ compEs2 es @ [Invoke M |es|]
compE2 {i:T; e} = compE2 e
compE2 (e1; e2) = compE2 e1 @ [Pop] @ compE2 e2
compE2 (if (e) e1 else e2) =
  (let cnd = compE2 e; thn = compE2 e1; els = compE2 e2; test = IfFalse (int (|thn| + 2));
   thnex = Goto (int (|els| + 1))
   in cnd @ [test] @ thn @ [thnex] @ els)
compE2 (while (e) c) =
  (let cnd = compE2 e; bdy = compE2 c; test = IfFalse (int (|bdy| + 3));
   loop = Goto (- int (|bdy| + |cnd| + 2))
   in cnd @ [test] @ bdy @ [Pop] @ [loop] @ [Push Unit])
compE2 (throw e) = compE2 e @ [Throw]
compE2 (try e1 catch (C i) e2) =
  (let catch = compE2 e2 in compE2 e1 @ [Goto (int |catch| + 2), Store i] @ catch)
compEs2 [] = []
compEs2 (e · es) = compE2 e @ compEs2 es

```

Fig. 33. Definition of $compE_2$

current program counter (by adding the size of the compiled code to the left to it) and the current stack size (by incrementing it suitably). Each try-catch generates one entry in the exception table. It contains the stack size d before execution of the try-catch: that is the stack size we must return to when an exception is encountered and execution continues with the catch block. The precise addresses in this exception table entry are of course implicitly determined by the layout of the code produced by $compE_2$. It is crucial that this entry is placed behind the exception tables belonging to the try and catch blocks: because the JVM searches exception tables from the left, it will only find this entry if none of the more directly enclosing handlers match. At least that is the intuition. The proof follows below.

5.4 Program Compilation

Lifting expression compilation to the level of programs is easy: simply replace all method bodies by their compiled version. This is defined generically:

```

compP :: ('a ⇒ 'b) ⇒ 'a prog ⇒ 'b prog
compP f ≡ map (compC f)

compC :: ('a ⇒ 'b) ⇒ 'a cdecl ⇒ 'b cdecl
compC f ≡ λ(C, D, Fdecls, Mdecls). (C, D, Fdecls, map (compM f) Mdecls)

compM :: ('a ⇒ 'b) ⇒ 'a mdecl ⇒ 'b mdecl
compM f ≡ λ(M, Ts, T, m). (M, Ts, T, f m)

```

Now we can lift the two compiler stages from expressions to programs:

```
compP1 :: J-prog ⇒ J1-prog
```

```

compxE2 (new  $C$ )  $pc$   $d$  = []
compxE2 (Cast  $C$   $e$ )  $pc$   $d$  = compxE2  $e$   $pc$   $d$ 
compxE2 (Val  $v$ )  $pc$   $d$  = []
compxE2 ( $e_1$   $\ll bop \gg$   $e_2$ )  $pc$   $d$  = compxE2  $e_1$   $pc$   $d$  @ compxE2  $e_2$  ( $pc$  + |compE2  $e_1$ |) ( $d$  + 1)
compxE2 (Var  $i$ )  $pc$   $d$  = []
compxE2 ( $i := e$ )  $pc$   $d$  = compxE2  $e$   $pc$   $d$ 
compxE2 ( $e.F\{D\}$ )  $pc$   $d$  = compxE2  $e$   $pc$   $d$ 
compxE2 ( $e_1.F\{D\} := e_2$ )  $pc$   $d$  = compxE2  $e_1$   $pc$   $d$  @ compxE2  $e_2$  ( $pc$  + |compE2  $e_1$ |) ( $d$  + 1)
compxE2 ( $e.M(es)$ )  $pc$   $d$  = compxE2  $e$   $pc$   $d$  @ compxEs2  $es$  ( $pc$  + |compE2  $e$ |) ( $d$  + 1)
compxE2  $\{i:T; e\}$   $pc$   $d$  = compxE2  $e$   $pc$   $d$ 
compxE2 ( $e_1; e_2$ )  $pc$   $d$  = compxE2  $e_1$   $pc$   $d$  @ compxE2  $e_2$  ( $pc$  + |compE2  $e_1$ | + 1)  $d$ 
compxE2 (if ( $e$ )  $e_1$  else  $e_2$ )  $pc$   $d$  =
(let  $pc_1 = pc$  + |compE2  $e$ | + 1;  $pc_2 = pc_1$  + |compE2  $e_1$ | + 1
 in compxE2  $e$   $pc$   $d$  @ compxE2  $e_1$   $pc_1$   $d$  @ compxE2  $e_2$   $pc_2$   $d$ )
compxE2 (while ( $b$ )  $e$ )  $pc$   $d$  = compxE2  $b$   $pc$   $d$  @ compxE2  $e$  ( $pc$  + |compE2  $b$ | + 1)  $d$ 
compxE2 (throw  $e$ )  $pc$   $d$  = compxE2  $e$   $pc$   $d$ 
compxE2 (try  $e_1$  catch ( $C$   $i$ )  $e_2$ )  $pc$   $d$  =
(let  $pc_1 = pc$  + |compE2  $e_1$ |
 in compxE2  $e_1$   $pc$   $d$  @ compxE2  $e_2$  ( $pc_1$  + 2)  $d$  @ [( $pc$ ,  $pc_1$ ,  $C$ ,  $pc_1$  + 1,  $d$ )]))
compEs2 []  $pc$   $d$  = []
compEs2 ( $e \cdot es$ )  $pc$   $d$  = compxE2  $e$   $pc$   $d$  @ compEs2  $es$  ( $pc$  + |compE2  $e$ |) ( $d$  + 1)
    
```

 Fig. 34. Definition of *compxE2*

$compP_1 \equiv compP (\lambda(pns, body). compE_1 (this \cdot pns) body)$

$compP_2 :: J_1\text{-prog} \Rightarrow jvm\text{-prog}$

$compP_2 \equiv compP compMb_2$

$compMb_2 \equiv$

$\lambda body. \text{let } ins = compE_2 body @ [\text{Return}]; xt = compxE_2 body 0 0$
 $\text{in } (max\text{-stack } body, max\text{-vars } body, ins, xt)$

Function $compP_1$ compiles each method body in the context of *this* and its parameter names as the only local variables. Function $compP_2$ compiles each method body into the required 4-tuple of maximal stack size (Fig. 35), number of local variables, instructions, and exception table.

The main compiler $J2JVM$ is simply the composition of $compP_1$ with $compP_2$.

5.5 Correctness of Stage 1

Although the translation from names to indices appears straightforward, formulating its correctness is already a bit involved. We want to prove preservation of the semantics, i.e. that an evaluation $P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle$ implies some evaluation $compP_1 P \vdash_1 \langle compE_1 Vs e, (h, ls) \rangle \Rightarrow \langle compE_1 Vs e', (h', ls') \rangle$. It is plausible that *set* Vs should be a superset of *fv* e and that $[Vs \mapsto] ls$ should be related to l . Equating them does not work because variables in Vs that have not yet been assigned to will not be in *dom* l — uninitialised variables again complicate matters. Hence we have to settle for the weaker $l \subseteq_m [Vs \mapsto] ls$ where

$m_1 \subseteq_m m_2 \equiv \forall a \in dom m_1. m_1 a = m_2 a$

is defined for arbitrary maps. This relationship should be preserved, i.e. $l' \subseteq_m [Vs \mapsto] ls'$ should hold.

We are now ready for the main theorem:

```

max-stack (new C) = 1
max-stack (Cast C e) = max-stack e
max-stack (Val v) = 1
max-stack (e1 <<bop>> e2) = max (max-stack e1) (max-stack e2) + 1
max-stack (Var i) = 1
max-stack (i := e) = max-stack e
max-stack (e.F{D}) = max-stack e
max-stack (e1.F{D} := e2) = max (max-stack e1) (max-stack e2) + 1
max-stack (e.M(es)) = max (max-stack e) (max-stacks es) + 1
max-stack {i:T; e} = max-stack e
max-stack (e1; e2) = max (max-stack e1) (max-stack e2)
max-stack (if (e) e1 else e2) = max (max-stack e) (max (max-stack e1) (max-stack e2))
max-stack (while (e) c) = max (max-stack e) (max-stack c)
max-stack (throw e) = max-stack e
max-stack (try e1 catch (C i) e2) = max (max-stack e1) (max-stack e2)
max-stacks [] = 0
max-stacks (e · es) = max (max-stack e) (1 + max-stacks es)

```

Fig. 35. Definition of *max-stack*

THEOREM 5.1. If *wwf-J-prog* P and $P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle$ and $fv\ e \subseteq set\ Vs$ and $l \subseteq_m [Vs \mapsto ls]$ and $|Vs| + max\ vars\ e \leq |ls|$ then $\exists ls'. compP_1\ P \vdash_1 \langle compE_1\ Vs\ e, (h, ls) \rangle \Rightarrow \langle fin_1\ e', (h', ls') \rangle \wedge l' \subseteq_m [Vs \mapsto ls']$.

First we examine the additional premises. Weak well-formedness is necessary for method calls: in order to apply the induction hypothesis arising from the evaluation of the method body, we need to establish an instance of $fv\ e \subseteq set\ Vs$, namely that the only free variables in the body are *this* and the parameters. The premise $|Vs| + max\ vars\ e \leq |ls|$ ensures that ls is large enough to hold all local variables needed during the evaluation of e . In the conclusion, $compE_1\ Vs\ e'$ has been replaced by $fin_1\ e'$ where $fin_1 :: expr \Rightarrow expr_1$ is the “identity” on final expressions: $fin_1\ (Val\ v) = Val\ v$ and $fin_1\ (Throw\ a) = Throw\ a$. This simplifies the proposition and the proof because it removes the pointless Vs .

PROOF. The proof is by induction on $P \vdash \langle e, (h, l) \rangle \Rightarrow \langle e', (h', l') \rangle$. The only non-trivial cases are blocks and try-catch. We discuss the block case as it isolates the main issue, local variables. Let e be $\{V:T; e_0\}$. From the assumptions $fv\ e \subseteq set\ Vs$, $l \subseteq_m [Vs \mapsto ls]$ and $|Vs| + max\ vars\ e \leq |ls|$ the preconditions for the induction hypothesis follow and we obtain an ls' such that $compP_1\ P \vdash_1 \langle compE_1\ Vs_0\ e_0, (h, ls) \rangle \Rightarrow \langle fin_1\ e', (h', ls') \rangle$ (1) and $l' \subseteq_m [Vs_0 \mapsto ls']$ (2), where Vs_0 is $Vs @ [V]$. This ls' is also the witness for our goal. Its first conjunct $compP_1\ P \vdash_1 \langle compE_1\ Vs\ e, (h, ls) \rangle \Rightarrow \langle fin_1\ e', (h', ls') \rangle$ follows directly from (1) by the rule for blocks. The second conjunct is the conclusion of the following general lemma about maps:

$$\begin{aligned} & \llbracket l \subseteq_m [Vs \mapsto ls]; l' \subseteq_m [Vs \mapsto ls', V \mapsto v]; V \in set\ Vs \implies ls'_{[index\ Vs\ V]} = ls'_{[index\ Vs\ V]}; \\ & \quad |ls| = |ls'|; |Vs| < |ls'| \rrbracket \\ & \implies l'(V := l\ V) \subseteq_m [Vs \mapsto ls'] \end{aligned}$$

It is proved via the chain $l'(V := l\ V) \subseteq_m [Vs \mapsto ls', V \mapsto v](V := l\ V) = [Vs \mapsto ls'](V := l\ V) \subseteq_m [Vs \mapsto ls']$ where the rightmost \subseteq_m is proved by a case distinction on whether $l\ V$ is *None* or not.

```

unmod (new C) i = True
unmod (Cast C e) i = unmod e i
unmod (Val v) i = True
unmod (e1 <<bop>> e2) i = (unmod e1 i ∧ unmod e2 i)
unmod (Var i) j = True
unmod (i := e) j = (i ≠ j ∧ unmod e j)
unmod (e.F{D}) i = unmod e i
unmod (e1.F{D} := e2) i = (unmod e1 i ∧ unmod e2 i)
unmod (e.M(es)) i = (unmod e i ∧ unmods es i)
unmod {j:T; e} i = unmod e i
unmod (e1; e2) i = (unmod e1 i ∧ unmod e2 i)
unmod (if (e) e1 else e2) i = (unmod e i ∧ unmod e1 i ∧ unmod e2 i)
unmod (while (e) c) i = (unmod e i ∧ unmod c i)
unmod (throw e) i = unmod e i
unmod (try e1 catch (C i) e2) j = (unmod e1 j ∧ (if i = j then False else unmod e2 j))
unmods [] i = True
unmods (e · es) i = (unmod e i ∧ unmods es i)
    
```

 Fig. 36. Definition of *unmod*

The first premise of the lemma is one of our assumptions. The second premise (with v being $ls'_{[|Vs|]}$) follows from (2) because $|Vs| < |ls|$, which follows from the assumption $|Vs| + \text{max-vars } e \leq |ls|$. Premise $|ls| = |ls'|$ holds because J_1 -evaluation does not change the length of the list of local variables. Thus we are left with the premise

$$V \in \text{set } Vs \implies ls'_{[\text{index } Vs \ V]} = ls'_{[\text{index } Vs \ V]}$$

which follows because evaluation does not modify hidden local variables. We call an element in a list **hidden** if it occurs again further to the right:

```

hidden :: 'a list ⇒ nat ⇒ bool
hidden xs i ≡ i < |xs| ∧ xs[i] ∈ set (drop (i + 1) xs)
    
```

Now we introduce a predicate $\text{unmod} :: \text{expr}_1 \Rightarrow \text{nat} \Rightarrow \text{bool}$ (Fig. 36) such that $\text{unmod } e \ i$ means i is not assigned to in e . It is easy to prove that hidden variables are unmodified and that evaluation preserves unmodified variables:

```

hidden Vs i ⇒ unmod (compE1 Vs e) i
[[P ⊢1 ⟨e, (h, ls)⟩ ⇒ ⟨e', (h', ls')⟩]; unmod e i; i < |ls|] ⇒ ls'[i] = ls'[i]
    
```

Now assume $V \in \text{set } Vs$. It follows easily that $\text{hidden } Vs_0 \ (\text{index } Vs \ V)$, hence that $\text{unmod} \ (\text{compE}_1 \ Vs_0 \ e) \ (\text{index } Vs \ V)$, and hence that $ls'_{[\text{index } Vs \ V]} = ls'_{[\text{index } Vs \ V]}$, thus proving the remaining premise. \square

5.6 Correctness of Stage 2

We need to prove that evaluation on the J_1 -level implies a related execution sequence on the JVM-level:

THEOREM 5.2. If $P_1 \vdash C$ sees $M: Ts \rightarrow T = \text{body in } C$ and $P_1 \vdash_1 \langle \text{body}, (h, ls) \rangle \Rightarrow \langle e', (h', ls') \rangle$ then $\text{compP}_2 \ P_1 \vdash (\text{None}, h, [([], ls, C, M, 0)]) \xrightarrow{\text{jvm}} (\text{exception } e', h', []).$

Essentially this theorem says that when executing a method body one obtains the same final heap on the JVM-level as on the J_1 -level. The local variables have disappeared from the final JVM state because we are right at the end of the computation, where the frame stack is empty. Function *exception* of type $'a \text{ exp} \Rightarrow \text{addr option}$ extracts the exception reference if there is one: $\text{exception } (\text{Throw } a) = [a]$ and $\text{exception } _ = \text{None}$.

Unfortunately, this theorem needs to be strengthened greatly before it becomes provable by induction. The result is

LEMMA 5.3. Let $P_1 :: J_1\text{-prog}$, $e :: \text{expr}_1$ and $P \equiv \text{comp}P_2 P_1$. Assume $P, C, M, pc \triangleright \text{comp}E_2 e$ and $P, C, M \triangleright \text{comp}xE_2 e \text{ pc } |vs| / I, |vs|$ and $\{pc..<pc + |\text{comp}E_2 e|\} \subseteq I$. If $P_1 \vdash_1 \langle e, (h, ls) \rangle \Rightarrow \langle e', (h', ls') \rangle$ then, letting $\sigma \equiv (\text{None}, h, (vs, ls), C, M, pc) \cdot fs$, there are two cases:

- If $e' = \text{Val } v$ then
 $P \vdash \sigma \xrightarrow{\text{jvm}} (\text{None}, h', (v \cdot vs, ls'), C, M, pc + |\text{comp}E_2 e|) \cdot fs$.
- If $e' = \text{Throw } a$ then there is a pc' such that $pc \leq pc' < pc + |\text{comp}E_2 e|$ and
 $\neg \text{caught } P \text{ pc}' h' a (\text{comp}xE_2 e \text{ pc } |vs|)$ and
 $P \vdash \sigma \xrightarrow{\text{jvm}} \text{find-handler } P \text{ a } h' ((vs, ls'), C, M, pc') \cdot fs$.

We will discuss the various components of this lemma in turn.

Instead of a method body we allow an arbitrary expression e . As a consequence we require that the pc in the top frame points to the compiled expression. To this end we define

$$P, C, M, pc \triangleright is \equiv is \leq \text{drop } pc (\text{instrs-of } P \ C \ M)$$

$$P, C, M, pc \triangleright i \equiv \exists is. \text{drop } pc (\text{instrs-of } P \ C \ M) = i \cdot is$$

where $P, C, M, pc \triangleright is$ means that the instruction list is is a prefix (\leq) of the instructions of method M starting at pc , and $P, C, M, pc \triangleright i$ means that i is the instruction at pc in M . Exception handling complicates matters further. We need to say that the exception table compiled from e is contained in the exception table of M in such a way that exceptions thrown by e are not erroneously caught by exception table entries further to the left (because exception tables are searched from the left). We also need to say that if an exception thrown by e is caught by an entry further to the right, that entry does not expect more elements on the stack than we currently have. This is what $P, C, M \triangleright \text{comp}xE_2 e \text{ pc } |vs| / I, |vs|$ expresses:

$$P, C, M \triangleright xt / I, d \equiv$$

$$\exists xt_0 \ xt_1.$$

$$\text{ex-table-of } P \ C \ M = xt_0 @ xt @ xt_1 \wedge \text{pcs } xt_0 \cap I = \{\} \wedge \text{pcs } xt \subseteq I \wedge$$

$$(\forall pc \in I. \forall C \text{ pc}' \ d'. \text{match-ex-table } P \ C \ pc \ xt_1 = [(pc', d')] \longrightarrow d' \leq d)$$

where $\text{ex-table-of } P \ C \ M$ returns the exception table component of *method* $P \ C \ M$, and $\text{pcs} :: \text{ex-table} \Rightarrow \text{pc set}$ yields all program counters guarded by some entry in the table: $\text{pcs } xt \equiv \bigcup_{(f, t, C, h, d) \in \text{set } xt} \{f..<t\}$. The requirement concerning the exception entries to the right of xt is expressed via function *match-ex-table* which we describe informally: $\text{match-ex-table } P \ C \ pc \ xt_1$ searches xt_1 for an entry matching an exception of class C thrown at pc . If it finds a suitable entry, it returns the corresponding pair (pc', d') of the handler pc and the size the stack has to be reduced to, which must be less or equal d .

Let us now turn to the conclusion of the lemma. If the J_1 evaluation ended in a value v , there is a corresponding JVM execution which deposits v on top of the operand stack. If the J_1 evaluation ended with an exception, there must be a pc' where the corresponding JVM exception is raised. This pc' must lie within the instruction list generated from e , the exception (identified by pc' and the class of the exception object $h' a$) must not be caught by an exception handler in e , and the final JVM state is the one reached by searching the frame stack for a matching handler. The definition of *caught* is straightforward and omitted.

PROOF. Lemma 5.3 is proved by induction on $P_1 \vdash_1 \langle e, (h, ls) \rangle \Rightarrow \langle e', (h', ls') \rangle$. Each case is a lengthy combination of JVM executions of the compiled subexpressions of e (obtained from the induction hypotheses) to obtain the JVM execution of $compE_2 e$. We will sketch one case, the penultimate rule in Fig. 4, which describes the evaluation of $e = \text{try } e_1 \text{ catch } (C' V) e_2$ when e_1 throws an exception of a subclass of C' . Let $\sigma_0 = (None, h_0, (vs, ls_0, C, M, pc) \cdot fs)$, $pc_1 = pc + |compE_2 e_1|$, and $\sigma_0' = (None, h_1, (Addr a \cdot vs, ls_1, C, M, pc_1 + 1) \cdot fs)$.

First we show $P \vdash \sigma_0 \xrightarrow{jvm} \sigma_0'$. From the induction hypothesis for the evaluation of e_1 we obtain pc' such that $pc \leq pc' < pc_1$ and $\neg \text{caught } P \ pc' \ h_1 \ a \ (compxE_2 \ e_1 \ pc \ |vs|)$ and $P \vdash \sigma_0 \xrightarrow{jvm} \text{find-handler } P \ a \ h_1 \ ((vs, ls_1, C, M, pc') \cdot fs)$. It remains to show that *find-handler* returns σ_0' , which is the case if it reaches the entry at the end of $xt = compxE_2 \ e \ pc \ |vs| = compxE_2 \ e_1 \ pc \ |vs| @ compxE_2 \ e_2 \ (pc_1 + 2) \ |vs| @ [(pc, pc_1, C, pc_1 + 1, |vs|)]$. From the assumption $P, C, M \triangleright xt / I, |vs|$ it follows that *ex-table-of* $P \ C \ M$ is of the form $xt_0 @ xt @ xt_1$ and that $pcs \ xt_0$ and I are disjoint. Because $pc \leq pc' < pc_1$ implies $pc' \in \{pc.. < pc + |compE_2 \ e|\}$ and because by assumption $\{pc.. < pc + |compE_2 \ e|\} \subseteq I$, we cannot have $pc' \in pcs \ xt_0$, i.e. the exception is not protected by an entry in xt_0 . We already know that it is not caught by $compxE_2 \ e_1 \ pc \ |vs|$ and it cannot be caught by $compxE_2 \ e_2 \ (pc_1 + 2) \ |vs|$ either because all of its entries protect program counters $\geq pc_1 + 2 > pc'$. Thus *find-handler* reaches the matching entry at the end of xt .

Executing the **Store** instruction yields $P \vdash \sigma_0' \xrightarrow{jvm} \sigma_1$ where $\sigma_1 = (None, h_1, (vs, ls_1[i := Addr a], C, M, pc_1 + 2) \cdot fs)$. Let $pc_2 = pc_1 + 2 + |compE_2 \ e_2|$. If the evaluation of e_2 ends in **Val** v it follows easily from the second induction hypothesis that $P \vdash \sigma_1 \xrightarrow{jvm} (None, h_2, (v \cdot vs, ls_2, C, M, pc_2) \cdot fs)$, as required. If it ends in an exception **Throw** xa , the second induction hypothesis yields a pc'' such that $pc_1 + 2 \leq pc'' < pc_2$ and $\neg \text{caught } P \ pc'' \ h_2 \ xa \ (compxE_2 \ e_2 \ (pc_1 + 2) \ |vs|)$ and $P \vdash \sigma_1 \xrightarrow{jvm} \sigma_2$ where $\sigma_2 = \text{find-handler } P \ xa \ h_2 \ ((vs, ls_2, C, M, pc'') \cdot fs)$. This pc'' will also be the witness of the overall goal. We need to show that $pc \leq pc'' < pc + |compE_2 \ e|$, which is trivial, that the exception is not caught by xt , which follows because it is not caught by $compxE_2 \ e_2 \ (pc_1 + 2) \ |vs|$ and the remaining entries of xt all protect program counters $< pc_1 < pc''$, and that $P \vdash \sigma_1 \xrightarrow{jvm} \sigma_2$, which we already have. \square

The proof of Theorem 5.2 from Lemma 5.3 is straightforward.

5.7 Main Correctness Theorem

Composing Theorems 5.1 and 5.2 yields the main correctness theorem:

THEOREM 5.4. If *wwf-J-prog* P and $P \vdash C$ sees $M: Ts \rightarrow T = (pns, body)$ in C and $P \vdash \langle body, (h, [this \cdot pns \mapsto vs]) \rangle \Rightarrow \langle e', (h', l') \rangle$ and $|vs| = |pns| + 1$ and $|rest| = \text{max-vars } body$ then $J2JVM P \vdash (None, h, ([], vs @ rest, C, M, 0)) \xrightarrow{jvm} (\text{exception } e', h', [])$.

If the program is weakly well-formed, then a big step evaluation of a Jinja method body implies a corresponding JVM execution of the compiled program. The initial JVM state contains the same heap, an empty operand stack, and a list of local variables consisting of the parameters values vs followed by an arbitrary $rest$ which only needs to be of the right length $\text{max-vars } body$.

One may also want to prove the converse: any result produced by the compiled program can be produced by the source program. Given the above correctness result and the fact that the JVM is deterministic, it would suffice to prove that if the compiled program terminates, so does the source program. We have not done so.

5.8 Preservation of Well-formedness: Stage 1

Now we turn from the semantics to the question of type correctness and general well-formedness of the generated code. Preservation of well-typedness by $\text{comp}E_1$ is easily proved by induction on e :

LEMMA 5.5. If $P, [Vs \mapsto Ts] \vdash e :: U$ and $|Ts| = |Vs|$ then $\text{comp}P f P, Ts \vdash_1 \text{comp}E_1 Vs e :: U$.

Because program compilation $\text{comp}P$ does not modify the types, which is reflected in a number of lemmas like $\text{comp}P f P \vdash T \leq T'$ iff $P \vdash T \leq T'$, it does not matter what f is.

The preservation of definite assignment requires a sequence of lemmas, all proved by induction on e , the first three automatically.

If $\mathcal{A} e = [A]$ then $A \subseteq \text{fv } e$.

If $\mathcal{A} e = \text{None}$ then $\mathcal{A} (\text{comp}E_1 Vs e) = \text{None}$.

$\mathcal{D} e = \text{None}$

If $\mathcal{A} e = [A]$ and $\text{fv } e \subseteq \text{set } Vs$ then $\mathcal{A} (\text{comp}E_1 Vs e) = [\text{index } Vs \text{ ' } A]$.

If $A \subseteq \text{set } Vs$ and $\text{fv } e \subseteq \text{set } Vs$ and $\mathcal{D} e [A]$ then $\mathcal{D} (\text{comp}E_1 Vs e) [\text{index } Vs \text{ ' } A]$.

Here $f \text{ ' } A$ is the image of A under f : $f \text{ ' } A \equiv \{y \mid \exists x \in A. y = f x\}$. The final lemma has the following corollary:

COROLLARY 5.6. If $\mathcal{D} e [\text{set } Vs]$ and $\text{fv } e \subseteq \text{set } Vs$ and *distinct* Vs then $\mathcal{D} (\text{comp}E_1 Vs e) [\{\cdot \cdot < |Vs|\}]$.

The combination of this corollary (applicable because all parameter names of a well-formed method are distinct and do not contain *this*), Lemma 5.5, and the easy lemma $\mathcal{B} (\text{comp}E_1 Vs e) |Vs|$ yields

THEOREM 5.7. If *wf-J-prog* P then *wf-J₁-prog* $(\text{comp}P_1 P)$.

The proof is straightforward using the lemma (where \bigwedge is universal quantification)

$$\begin{aligned}
& \llbracket \bigwedge C M Ts T m. \\
& \quad \llbracket P \vdash C \text{ sees } M: Ts \rightarrow T = m \text{ in } C; wf\text{-mdecl } wf_1 P C (M, Ts, T, m) \rrbracket \\
& \quad \implies wf\text{-mdecl } wf_2 (compP f P) C (M, Ts, T, f m); \\
& \quad wf\text{-prog } wf_1 P \rrbracket \\
& \implies wf\text{-prog } wf_2 (compP f P)
\end{aligned}$$

which essentially says that $compP f$ turns a wf_1 well-formed program into a wf_2 well-formed program if f turns a wf_1 well-formed method into a wf_2 well-formed method.

5.9 Preservation of Well-formedness: Stage 2

We have to show that $compE_2$ maps a well-typed J_1 expression into a well-typed instruction list. For that purpose we follow Stärk et al. [2001] and define a “type compiler” that compiles an expression into type annotations (a method type) for the instruction list that $compE_2$ produces:

$$compT :: ty \text{ list} \Rightarrow nat \text{ set option} \Rightarrow ty \text{ list} \Rightarrow expr_1 \Rightarrow ty_i' \text{ list}$$

As in §4.7 we fix the program and the method we are in:

$$\begin{aligned}
P & :: J_1\text{-prog} && \text{the program,} \\
mxl & :: nat && \text{number of local variables,} \\
mxs & :: nat && \text{maximum stack size,} \\
T_r & :: ty && \text{and return type of the method.}
\end{aligned}$$

The type compiler call $compT E A ST e$ produces the method type for $compE_2 e$ in the context of the environment E assuming that initially a) the local variables in $A :: nat \text{ set option}$ are initialised and b) the stack elements are typed according to $ST :: ty \text{ list}$. It uses the following auxiliary functions:

$$ty_l E A' \equiv map (\lambda i. \text{if } i \in A' \wedge i < |E| \text{ then } OK E_{[i]} \text{ else } Err) [0..<mxl]$$

computes the typing of the local variables based on their declared types in E and their initialisation status in $A' :: nat \text{ set}$.

$$ty_i' ST E A \equiv \text{case } A \text{ of } None \Rightarrow None \mid [A'] \Rightarrow [(ST, ty_l E A)]$$

computes the state type described by ST , E and A . Remember that $A = None$ indicates unreachability.

$$\text{after } E A ST e \equiv ty_i' (ty E e \cdot ST) E (A \sqcup A e)$$

produces the state type characterising stack and local variables after the evaluation of e . Thus we need to push the type of e onto the stack and take the effect of e on A into account. Function ty is a functional version of type checking and is characterised by $P, E \vdash_1 e :: T \implies ty E e = T$

The full definition of $compT$ is shown in Fig. 37 and uses the additional function

$$compT_a E A ST e \equiv compT E A ST e @ [\text{after } E A ST e]$$

because the method type that $compT$ produces (intentionally) lacks the state types directly before and after the execution of the corresponding instruction list. That is, it is one element shorter than the instruction list, describing only the states between the instructions. As a simple example let e be the expression $1 := Val (Intg 42)$ and hence

$$\begin{aligned}
\text{compT } E \ A \ ST \ (\text{new } C) &= [] \\
\text{compT } E \ A \ ST \ (\text{Cast } C \ e) &= \text{compT}_a \ E \ A \ ST \ e \\
\text{compT } E \ A \ ST \ (\text{Val } v) &= [] \\
\text{compT } E \ A \ ST \ (e_1 \ll\text{bop}\gg e_2) &= \\
(\text{let } ST_1 = \text{ty } E \ e_1 \cdot ST; A_1 = A \sqcup \mathcal{A} \ e_1 \text{ in } \text{compT}_a \ E \ A \ ST \ e_1 \ @ \ \text{compT}_a \ E \ A_1 \ ST_1 \ e_2) &= \\
\text{compT } E \ A \ ST \ (\text{Var } i) &= [] \\
\text{compT } E \ A \ ST \ (i := e) &= \text{compT}_a \ E \ A \ ST \ e \ @ \ [\text{ty}_i' \ ST \ E \ (A \sqcup \mathcal{A} \ e \sqcup \{\{i\}\})] \\
\text{compT } E \ A \ ST \ (e.F\{D\}) &= \text{compT}_a \ E \ A \ ST \ e \\
\text{compT } E \ A \ ST \ (e_1.F\{D\} := e_2) &= \\
(\text{let } ST_1 = \text{ty } E \ e_1 \cdot ST; A_1 = A \sqcup \mathcal{A} \ e_1; A_2 = A_1 \sqcup \mathcal{A} \ e_2 & \\
\text{in } \text{compT}_a \ E \ A \ ST \ e_1 \ @ \ \text{compT}_a \ E \ A_1 \ ST_1 \ e_2 \ @ \ [\text{ty}_i' \ ST \ E \ A_2]) & \\
\text{compT } E \ A \ ST \ \{i:T; e\} &= \text{compT} \ (E \ @ \ [T]) \ (A \ominus i) \ ST \ e \\
\text{compT } E \ A \ ST \ (e_1; e_2) &= \\
(\text{let } A_1 = A \sqcup \mathcal{A} \ e_1 \text{ in } \text{compT}_a \ E \ A \ ST \ e_1 \ @ \ [\text{ty}_i' \ ST \ E \ A_1] \ @ \ \text{compT} \ E \ A_1 \ ST \ e_2) & \\
\text{compT } E \ A \ ST \ (\text{if } (e) \ e_1 \ \text{else } e_2) &= \\
(\text{let } A_0 = A \sqcup \mathcal{A} \ e; \tau = \text{ty}_i' \ ST \ E \ A_0 & \\
\text{in } \text{compT}_a \ E \ A \ ST \ e \ @ \ [\tau] \ @ \ \text{compT}_a \ E \ A_0 \ ST \ e_1 \ @ \ [\tau] \ @ \ \text{compT} \ E \ A_0 \ ST \ e_2) & \\
\text{compT } E \ A \ ST \ (\text{while } (e) \ c) &= \\
(\text{let } A_0 = A \sqcup \mathcal{A} \ e; A_1 = A_0 \sqcup \mathcal{A} \ c; \tau = \text{ty}_i' \ ST \ E \ A_0 & \\
\text{in } \text{compT}_a \ E \ A \ ST \ e \ @ \ [\tau] \ @ \ \text{compT}_a \ E \ A_0 \ ST \ c \ @ \ [\text{ty}_i' \ ST \ E \ A_1, \text{ty}_i' \ ST \ E \ A_0]) & \\
\text{compT } E \ A \ ST \ (\text{throw } e) &= \text{compT}_a \ E \ A \ ST \ e \\
\text{compT } E \ A \ ST \ (e.M(es)) &= \text{compT}_a \ E \ A \ ST \ e \ @ \ \text{compTs} \ E \ (A \sqcup \mathcal{A} \ e) \ (\text{ty } E \ e \cdot ST) \ es \\
\text{compT } E \ A \ ST \ (\text{try } e_1 \ \text{catch } (C \ i) \ e_2) &= \text{compT}_a \ E \ A \ ST \ e_1 \ @ \\
[\text{ty}_i' \ (\text{Class } C \cdot ST) \ E \ A, \text{ty}_i' \ ST \ (E \ @ \ [\text{Class } C]) \ (A \sqcup \{\{i\}\})] \ @ & \\
\text{compT} \ (E \ @ \ [\text{Class } C]) \ (A \sqcup \{\{i\}\}) \ ST \ e_2 & \\
\text{compTs } E \ A \ ST \ [] &= [] \\
\text{compTs } E \ A \ ST \ (e \cdot es) &= \text{compT}_a \ E \ A \ ST \ e \ @ \ \text{compTs} \ E \ (A \sqcup \mathcal{A} \ e) \ (\text{ty } E \ e \cdot ST) \ es
\end{aligned}$$

Fig. 37. Type compiler

$$\text{compE}_2 \ e = [\text{Push} \ (\text{Intg } 42), \text{Store } 1, \text{Push } \text{Unit}]$$

With $E = [\text{Class } C, \text{Integer}]$, $A = [\{0\}]$ and $ST = []$ we obtain

$$\text{compT } E \ A \ ST \ e = [[([\text{Integer}], [\text{OK} \ (\text{Class } C), \text{Err}]), [([], [\text{OK} \ (\text{Class } C), \text{OK} \ \text{Integer}]])]$$

Since we are dealing with instruction lists in isolation rather than whole methods, we need a new well-typedness notion. We write $\vdash is, xt \ [\:] \ \tau s$ to mean that the instruction list is together with the exception table xt is well-typed w.r.t. the method type $\tau s \ :: \ \text{ty}_i' \ \text{list}$:

$$\begin{aligned}
\vdash is, xt \ [\:] \ \tau s &\equiv \\
|is| < |\tau s| \wedge \text{pcs } xt \subseteq \{0..<|is|\} \wedge (\forall pc < |is|. P, Tr, mxs, |\tau s|, xt \vdash is_{[pc], pc} \ :: \ \tau s) &
\end{aligned}$$

The key theorem says that the instruction list and exception table are well-typed w.r.t. the method type produced by the type compiler:

THEOREM 5.8. If $P, E \vdash_1 e \ :: \ T$ and $\mathcal{D} \ e \ A$ and $\mathcal{B} \ e \ |E|$ and $|ST| + \text{max-stack } e \leq \text{mxs}$ and $|E| + \text{max-vars } e \leq \text{mxl}$ then $\vdash \text{compE}_2 \ e, \text{compxE}_2 \ e \ 0 \ |ST| \ [\:] \ \text{ty}_i' \ ST \ E \ A \cdot \text{compT}_a \ E \ A \ ST \ e$.

$\vdash [], [] [::] \tau_s$	
$\vdash is_e, xt_e [::] \tau \cdot \tau_{s_e} @ [\tau_e]$	Ind.hyp.
$\vdash is_e, xt_e [::] \tau_s$	Lemma 5.9
$\vdash [test], [] [::] \tau_e \cdot \tau_1 \cdot \tau_{s_c} @ [\tau_c, \tau_2, \tau_1, \tau']$	
$\vdash is_e @ [test], xt_e [::] \tau_s$	Lemma 5.9
$\vdash is_c, xt_c [::] \tau_1 \cdot \tau_{s_c} @ [\tau_c]$	Ind.hyp.
$\vdash is_e @ [test] @ is_c, xt_e @ xt_c' [::] \tau_s$	Lemma 5.9
$\vdash [Pop], [] [::] [\tau_c, \tau_2]$	
$\vdash is_e @ [test] @ is_c @ [Pop], xt_e @ xt_c' [::] \tau_s$	Lemma 5.9
$P \vdash \tau_2 \leq' \tau$	
$P, T_r, mxs, \tau_s , [] \vdash goto, is_e + is_c + 2 :: \tau_s$	
$\vdash is_e @ [test] @ is_c @ [Pop, goto], xt_e @ xt_c' [::] \tau_s$	Lemma 5.10
$\vdash [Push Unit], [] [::] [\tau_1, \tau']$	
$\vdash compE_2(\text{while}(e) c), compxE_2(\text{while}(e) c) 0 ST [::] \tau_s$	Lemma 5.9

 Fig. 38. Proof summary for **while**

PROOF. By induction on e . In order to combine different well-typedness propositions we need a central lemma which follows readily from the definitions:

LEMMA 5.9. If $\vdash is_1, xt_1 [::] \tau_{s_1} @ \tau_{s_2}$ and $\vdash is_2, xt_2 [::] \tau_{s_3}$ and $|\tau_{s_1}| = |is_1|$ and $\tau_{s_3} \leq \tau_{s_2}$ then $\vdash is_1 @ is_2, xt_1 @ shift |is_1| xt_2 [::] \tau_{s_1} @ \tau_{s_2}$.

Here $shift\ n\ xt$ shifts all program counters in xt by n and \leq again means prefix. This lemma works well in the presence of forward jumps, but for backward ones we need a second one:

LEMMA 5.10. If $\vdash is, xt [::] \tau_s$ and $P, T_r, mxs, mpc, [] \vdash i, pc :: \tau_s$ and $pc = |is|$ and $mpc = |\tau_s|$ and $|is| + 1 < |\tau_s|$ then $\vdash is @ [i], xt [::] \tau_s$.

We present just one case of the proof of Theorem 5.8, the while-loop. The correspondence of instructions $compE_2(\text{while}(e) c)$ (first column) and types $compTEAST(\text{while}(e) c)$ (second column) is best described by a table:

$compE_2\ e$	$\tau \cdot \tau_{s_e}$	where $\tau = ty_i' ST E A$ and $\tau_{s_e} = compTEAST\ e$
IfFalse ...	τ_e	where $\tau_e = ty_i' (Boolean \cdot ST) E A_0$ and $A_0 = A \sqcup A$
$compE_2\ c$	$\tau_1 \cdot \tau_{s_c}$	where $\tau_1 = ty_i' ST E A_0$ and $\tau_{s_c} = compTEAST\ c$
Pop	τ_c	where $\tau_c = ty_i' (ty\ E\ c \cdot ST) E A_1$ and $A_1 = A_0 \sqcup A\ c$
Goto ...	τ_2	where $\tau_2 = ty_i' ST E A_1$
Push Unit	τ_1	
	τ'	where $\tau' = ty_i' (Void \cdot ST) E A_0$

Using the abbreviations $is_e = compE_2\ e$, $xt_e = compxE_2\ e\ 0\ |ST|$, $is_c = compE_2\ c$, $xt_c = compxE_2\ c\ 0\ |ST|$, $xt_c' = shift(|is_e| + 1) xt_c$, $test = \text{IfFalse} \dots$, $goto = \text{Goto} \dots$ and $\tau_s = \tau \cdot \tau_{s_e} @ [\tau_e, \tau_1] @ \tau_{s_c} @ [\tau_c, \tau_2, \tau_1, \tau']$ (i.e. $\tau_s = compTEAST(\text{while}(e) c)$), we can summarize the proof in the table in Fig. 38 where the uncommented lines follow easily.

The remaining cases follow the same pattern. Try-catch requires

LEMMA 5.11. If $|is_1| = |\tau_{s_1}|$ and $is\text{-class}\ P\ C$ and $|ST| < mxs$ and $\vdash is_1 @ is_2, xt [::] \tau_{s_1} @ (ty_i' (Class\ C \cdot ST) E A \cdot \tau_{s_2})$ and

$$\begin{aligned} & \forall \tau \in \text{set } \tau s_1. \\ & \quad \forall ST' LT'. \\ & \quad \tau = \lfloor (ST', LT') \rfloor \longrightarrow \\ & \quad \lfloor ST \rfloor \leq \lfloor ST' \rfloor \wedge P \vdash \lfloor (\text{drop } (|ST'| - |ST|) ST', LT') \rfloor \leq' ty_i' ST EA \end{aligned}$$

then $\vdash is_1 @ is_2, xt @ [(0, |is_1| - 1, C, |is_1|, |ST|)] [::] \tau s_1 @ (ty_i' (Class C \cdot ST) EA \cdot \tau s_2)$.

to justify the additional exception table entry. \square

Now it is not very difficult to conclude

THEOREM 5.12. If $wf\text{-}J_1\text{-prog } P$ then $wf\text{-}jvm\text{-prog } (compP_2 P)$.

By definition of $wf\text{-}jvm\text{-prog}$ we need a table Φ such that $wf\text{-}jvm\text{-prog}_\Phi (compP_2 P)$. The table can be computed by $compTP :: J_1\text{-prog} \Rightarrow ty_P$

$$\begin{aligned} & compTP P C M \equiv \\ & \text{let } (D, Ts, T, e) = \text{method } P C M; E = \text{Class } C \cdot Ts; A = \{\cdot | Ts \}; \\ & \quad mxl = 1 + |Ts| + \text{max-vars } e \\ & \text{in } ty_i' mxl [] EA \cdot compT_a P mxl EA [] e \end{aligned}$$

and the proof of $wf\text{-}jvm\text{-prog}_\Phi$ follows largely from Theorem 5.8. Note that ty_i' and $compT_a$ are explicitly supplied with their implicit arguments mxl and P because those are no longer fixed globally but are the specific values in this definition.

5.10 Preservation of Well-formedness: Main theorem

Combining Theorems 5.7 and 5.12 yields the main theorem:

THEOREM 5.13. If $wf\text{-}J\text{-prog } P$ then $wf\text{-}jvm\text{-prog } (J2JVM P)$.

If the source program is wellformed—which includes every method body being well-typed—then the compiled program is also wellformed and will pass bytecode verification. This is a non-trivial property not guaranteed by Java compilers (for example JDK 1.2 and 1.3) [Stärk and Schmid 2001].

5.11 Related Work

There is a sizable amount of literature on compiler verification in general [Dave 2003], which we cannot possibly survey here, but very little specifically for Java. An early landmark in mechanically verified compilers is the work by Young [1989] whose source language has procedures but no OO features, but whose target language is a real rather than a virtual machine. The work by Strecker [2002] and Klein and Strecker [2004] is closely related to ours but their main theorem does not talk about exceptions, which is a considerable simplification. Furthermore, their type compiler intrinsically assumes that variables are initialized and do not change their type (which they can if you have local variables and reuse storage as we do).

Stärk et al. [2001] show that terminating computations are compiled into terminating ones, just as we have done. They also claim to prove a 1:1 correspondence between Java and JVM executions. However, their main theorem fails to imply that nonterminating computations are compiled into nonterminating ones (which we have not shown either). Its statement and proof need to be augmented to prove that one cannot have a diverging Java computation in correspondence with a terminating JVM execution. Furthermore, it is interesting to note that their compiler has one stage only. Our stage 1 disappears in the sentence “we suppress the details

of a consistent assignment of JVM variable numbers \bar{x} to (occurrences of) Java variables x ” combined with the fact that their Java semantics treats local variables like the JVM: upon exit of a block, the local variables of the block are unchanged and still visible. This works only because Java forbids nested declarations of the same variable, which Jinja allows. Finally it is interesting to note that their formalization uses attributed syntax trees that tell you, for example, what the current environment E of each subexpression is, whereas we pass E explicitly into many of our functions. Although this is true for the whole formalization, it is most noticeable for the type compiler with its lengthy parameter list. It may well reduce clutter in our formalization to work with expressions annotated with E etc. We have not done so to minimize the number of concepts needed.

The correctness of compiling exceptions is studied by Hutton and Wright [2004]. They treat a very simple expression language and a stack machine where all exception handling information is kept on the stack. Because there is no separate exception table, this leads to simpler exception handling on the machine level and simpler proofs. It is possible that their machine could serve as a convenient stepping stone between Jinja and the Jinja VM.

League et al. [2002] compile Featherweight Java into F_ω , which is very different from our compilation into the JVM.

6. CONCLUSION

We have given a completely formal account of the core of a Java-like sequential language, abstract machine and compiler. Although there are some interesting contributions to the analysis of individual facets, for example definite assignment, the emphasis is on a unified model. At the same time we have demonstrated that it is possible to present this model in a style appropriate for a scientific journal, although the formal and machine-checked meta-language prohibits some of the liberties (like “. . .”) of traditional journal articles (and we have decided not to take others like hiding injections such as `[.]`). Of course Jinja is still a small language compared to Java, but we hope that our theories will become the basis for further extensions of Jinja, just as others, e.g. Büchi and Weck [1998], have extended some of our earlier Java formalisations.

The whole development (excluding this paper) runs to 20000 lines of Isabelle/HOL text (with few comments), roughly 350 printed pages and just over 1000 theorems (available at www.in.tum.de/~nipkow/Jinja/). The proofs take about 25min to process on a 3GHz Pentium IV with 1GB RAM. This is a major investment, but one that can be built up to gradually: from type checking via prototyping to full-blown theorem proving. The early stages are cheap and worthwhile for any language definition, the final stage is more of an investment but yields a considerable increase in confidence—in particular since the proofs are structured and the general drift of an argument can be followed even by a non-theorem-proving-expert.

We have already hinted that the whole formalisation is executable. (With the exception of *wf-jvm-prog*, a specification that is implemented by the executable BV.) That is, Isabelle/HOL supports the translation of a mixture of recursive functions and inductive relations into executable (ML) code [Berghofer and Nipkow 2002; Berghofer 2003] and we have run test cases to increase our confidence in the intu-

itive correctness of our definitions. Although we have not gone into this aspect at all, we consider prototyping essential for projects like this, where the initial specification (here: the source language) is already too large to be “obviously” correct. In addition we consider these executable ML prototypes as reference implementations for the language, the compiler, the bytecode verifier etc. Although these prototypes do not have the performance required for production purposes, theorem provers offer the right environment for achieving the required performance: efficient programs can be proved correct w.r.t. initial specifications and then exported to ML just like the prototypes. In fact, the Jinja code generator is a standard recursive functional program without particular performance bottlenecks and may not need much tuning. The bytecode verifier is another matter because its fixed point engine passes a large list around in a single-threaded manner but the generated ML code does not take advantage of this. ACL2 and PVS have code generators that could safely implement this list by an array that is updated destructively. There is ongoing work to provide similar features in Isabelle. We are convinced that theorem provers provide the right means to develop certified efficient language processors that are largely functional but imperative where necessary. Liu and Moore [2003] already provide strong evidence for this thesis.

ACKNOWLEDGMENTS

This work is directly based on and would not exist without the Isabelle formalisations of various facets of Java by David von Oheimb, Conny Pusch, Norbert Schirmer, Martin Strecker and Martin Wildmoser, for which we are very grateful. We also thank them and the referees for commenting on draft versions of this paper.

REFERENCES

- ALVES-FOSS, J., Ed. 1999. *Formal Syntax and Semantics of Java*. LNCS, vol. 1523. Springer-Verlag.
- ANCONA, D., LAGORIO, G., AND ZUCCA, E. 2001. A core calculus for Java exceptions. In *Proc. 16th ACM Conf. Object oriented programming, systems, languages, and applications*. 16–30.
- ASPINALL, D. 2000. Proof General — a generic tool for proof development. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS 2000*, S. Graf and M. Schwartzbach, Eds. LNCS, vol. 1785. Springer-Verlag, 38–42.
- BALLARIN, C. 2003. Locales and locale expressions in Isabelle/Isar. In *Types for Proofs and Programs, TYPES 2003*, S. Berardi, M. Coppo, and F. Damiani, Eds. LNCS, vol. 3085. Springer-Verlag, 34–50.
- BARTHE, G. AND DUFAY, G. 2004. A tool-assisted framework for certified bytecode verification. In *Fundamental Approaches to Software Engineering, FASE 2004*, M. Wermelinger and T. Margaria, Eds. LNCS, vol. 2984. Springer-Verlag, 99–113.
- BARTHE, G., DUFAY, G., JAKUBIEC, L., SERPETTE, B., AND DE SOUSA, S. M. 2001. A formal executable semantics of the JavaCard platform. In *Programming Languages and Systems (ESOP 2001)*, D. Sands, Ed. LNCS, vol. 2028. Springer-Verlag, 302–319.
- BERGHOFER, S. 2003. Proofs, programs and executable specifications in higher order logic. Ph.D. thesis, Institut für Informatik, Technische Universität München.
- BERGHOFER, S. AND NIPKOW, T. 2002. Executing higher order logic. In *Types for Proofs and Programs (TYPES’00)*, P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, Eds. LNCS, vol. 2277. Springer-Verlag, 24–40.
- BERTELSEN, P. 1997. Semantics of Java bytecode. Tech. rep., Technical University of Denmark. Mar. <http://home.tiscali.dk/petermb/>.

- BÜCHI, M. AND WECK, W. 1998. Compound types for Java. In *Proc. 13th ACM conf. Object-oriented programming, systems, languages, and application*. ACM Press.
- COGLIO, A. 2004. Simple verification technique for complex Java bytecode subroutines. *Concurrency and Computation: Practice and Experience* 16, 7 (June), 647–670.
- COGLIO, A., GOLDBERG, A., AND QIAN, Z. 2000. Toward a provably-correct implementation of the JVM bytecode verifier. In *Proc. DARPA Information Survivability Conference and Exposition (DISCEX'00), Vol. 2*. IEEE Computer Society Press, 403–410.
- COHEN, R. 1997. The defensive Java virtual machine specification. Tech. rep., Computational Logic Inc. <http://www.cli.com/software/djvm/>.
- DAVE, M. A. 2003. Compiler verification: A bibliography. *SIGSOFT Softw. Eng. Notes* 28, 6, 2–2.
- DROSSOPOULOU, S. AND EISENBACH, S. 1999. Describing the semantics of Java and proving type soundness. See Alves-Foss [1999], 41–82.
- FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1999. A programmer's reduction semantics for classes and mixins. See Alves-Foss [1999], 241–269.
- FREUND, S. N. 2000. Type systems for object-oriented intermediate languages. Ph.D. thesis, Stanford University.
- FREUND, S. N. AND MITCHELL, J. C. 2003. A type system for the Java bytecode language and verifier. *J. Automated Reasoning* 30, 271–321.
- GOLDBERG, A. 1998. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conf. Computer and Communications Security*. ACM Press, 49–58.
- HARTEL, P. AND MOREAU, L. 2001. Formalizing the safety of Java, the Java virtual machine and Java card. *ACM Comput. Surv.* 33, 517–558.
- HUISMAN, M. 2001. Reasoning about Java programs in higher order logic with PVS and Isabelle. Ph.D. thesis, Universiteit Nijmegen.
- HUTTON, G. AND WRIGHT, J. 2004. Compiling exceptions correctly. In *Mathematics of Program Construction, MPC 2004*, D. Kozen and C. Shankland, Eds. LNCS, vol. 3125. Springer-Verlag, 211–227.
- IGARASHI, A., PIERCE, B. C., AND WADLER, P. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 396–450.
- KILDALL, G. A. 1973. A unified approach to global program optimization. In *Proc. ACM Symp. Principles of Programming Languages*. ACM Press, 194–206.
- KLEIN, G. 2003. Verified Java bytecode verification. Ph.D. thesis, Institut für Informatik, Technische Universität München.
- KLEIN, G. AND NIPKOW, T. 2001. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience* 13, 1133–1151.
- KLEIN, G. AND NIPKOW, T. 2003. Verified bytecode verifiers. *Theor. Comput. Sci.* 298, 583–626.
- KLEIN, G. AND STRECKER, M. 2004. Verified Bytecode Verification and type-certifying Compilation. *Journal of Logic and Algebraic Programming* 58, 1–2, 27–60.
- KLEIN, G. AND WILDMOSER, M. 2003. Verified bytecode subroutines. *J. Automated Reasoning* 30, 363–398.
- LEAGUE, C., SHAO, Z., AND TRIFONOV, V. 2002. Type-preserving compilation of Featherweight Java. *ACM Trans. Program. Lang. Syst.* 24, 112–152.
- LEROY, X. 2003. Java bytecode verification: Algorithms and formalizations. *J. Automated Reasoning* 30, 235–269.
- LINDHOLM, T. AND YELLIN, F. 1999. *The Java Virtual Machine Specification*. Addison-Wesley.
- LIU, H. AND MOORE, J. S. 2003. Executable JVM model for analytical reasoning: a study. In *Proc. 2003 Workshop Interpreters, Virtual Machines and Emulators*. ACM Press, 15–23.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- NIPKOW, T. 1991. Higher-order critical pairs. In *6th IEEE Symp. Logic in Computer Science*. IEEE Computer Society Press, 342–349.
- NIPKOW, T. 2001. Verified bytecode verifiers. In *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, F. Honsell, Ed. LNCS, vol. 2030. Springer-Verlag, 347–363.

- NIPKOW, T., Ed. 2003a. *Special Issue on Java Bytecode Verification*. *J. Automated Reasoning*, vol. 30, 3–4.
- NIPKOW, T. 2003b. Structured Proofs in Isar/HOL. In *Types for Proofs and Programs (TYPES 2002)*, H. Geuvers and F. Wiedijk, Eds. LNCS, vol. 2646. Springer-Verlag, 259–278.
- NIPKOW, T. 2005. Jinja: Towards a comprehensive formal semantics for a Java-like language. In *Proof Technology and Computation, Proc. Marktoberdorf Summer School 2003*, H. Schwichtenberg and K. Spies, Eds. IOS Press.
- NIPKOW, T. AND OHEIMB, D. V. 1998. Java_{light} is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, 161–170.
- NIPKOW, T., PAULSON, L., AND WENZEL, M. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS, vol. 2283. Springer-Verlag. <http://www.in.tum.de/~nipkow/LNCS2283/>.
- OHEIMB, D. V. AND NIPKOW, T. 1999. Machine-checking the Java specification: Proving type-safety. See Alves-Foss [1999], 119–156.
- PUSCH, C. 1999. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, W. Cleaveland, Ed. LNCS, vol. 1579. Springer-Verlag, 89–103.
- QIAN, Z. 2000. Standard fixpoint iteration for Java bytecode verification. *ACM Trans. Program. Lang. Syst.* 22, 638–672.
- ROSE, E. 2002. Vérification de Code d’Octet de la Machine Virtuelle Java. Formalisation et Implantation. Ph.D. thesis, Université Paris VII.
- ROSE, E. 2003. Lightweight bytecode verification. *J. Automated Reasoning* 31, 3–4, 303–334.
- ROSE, E. AND ROSE, K. 1998. Lightweight bytecode verification. In *OOPSLA '98 Workshop Formal Underpinnings of Java*.
- SCHIRMER, N. 2003. Java Definite Assignment in Isabelle/HOL. In *Formal Techniques for Java-like Programs 2003 (Proceedings)*, S. Eisenbach, G. Leavens, P. Müller, A. Poetzsch-Heffter, and E. Poll, Eds. Chair of Software Engineering, ETH Zürich. Technical Report 108.
- SCHIRMER, N. 2004. Analysing the Java package/access concepts in Isabelle/HOL. *Concurrency and Computation: Practice and Experience* 16, 689–706.
- STÄRK, R. AND SCHMID, J. 2001. The problem of bytecode verification in current implementations of the JVM. Tech. rep., Department of Computer Science, ETH Zürich.
- STÄRK, R., SCHMID, J., AND BÖRGER, E. 2001. *Java and the Java Virtual Machine — Definition, Verification, Validation*. Springer-Verlag.
- STATA, R. AND ABADI, M. 1998. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, 149–161.
- STRECKER, M. 2002. Formal verification of a Java compiler in Isabelle. In *Automated Deduction — CADE-18*, A. Voronkov, Ed. LNCS, vol. 2392. Springer-Verlag, 63–77.
- SYME, D. 1999. Proving Java type soundness. See Alves-Foss [1999], 83–118.
- WENZEL, M. 2002. Isabelle/Isar — a versatile environment for human-readable formal proof documents. Ph.D. thesis, Institut für Informatik, Technische Universität München. <http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html>.
- WRIGHT, A. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Information and Computation* 115, 38–94.
- YOUNG, W. D. 1989. A mechanically verified code generator. *J. Automated Reasoning* 5, 493–518.

Index

- $::$, 2
- \Rightarrow , 2
- $\llbracket \cdot \rrbracket \Rightarrow$, 3
- $\{ \cdot \cdot \}$, 3
- $\{ \cdot \cdot < \}$, 3
- $\{ \cdot \cdot \}$, 3
- $\{ \cdot \cdot < \}$, 3
- $\lfloor \cdot \rfloor$, 3
- $\textcircled{\cdot}$, 3
- $\lfloor \cdot \rfloor$, 3
- $\lfloor \cdot \cdot < \rfloor$, 3
- $\lfloor \cdot \rfloor$, 3
- \cdot , 3
- $\lfloor \cdot \rfloor$, 3
- $\lfloor \lfloor \cdot \rfloor \rfloor$, 3
- $\lfloor \cdot \rfloor$, 3
- $(\lfloor \cdot \rfloor)$, 3
- $(:=)$, 3
- (\mapsto) , 3
- $++$, 3
- \mapsto , 3
- $\cdot \{ \cdot \} :=$, 5
- $\cdot \{ \cdot \}$, 5
- $\cdot ()$, 5
- $\ll \gg$, 5
- $+$, 5
- $:=$, 5
- $;$, 5
- $=$, 5
- $\vdash \textit{has} : \textit{in}$, 6
- $\vdash \textit{has-fields}$, 7
- $\vdash \textit{sees} : \textit{in}$, 6
- $\vdash \textit{sees} : \mapsto = \textit{in}$, 6
- $\vdash \prec^*$, 6
- $\vdash \prec^1$, 7
- $\vdash \langle \cdot \rangle \Rightarrow \langle \cdot \rangle$, 8
- $\vdash \langle \cdot \rangle \rightarrow \langle \cdot \rangle$, 13
- $\vdash \langle \cdot \rangle \mapsto \langle \cdot \rangle$, 14
- $\vdash \langle \cdot \rangle \rightarrow^* \langle \cdot \rangle$, 16
- $\vdash \langle \cdot \rangle \mapsto^* \langle \cdot \rangle$, 16
- $\vdash \leq$, 21
- $\vdash \lfloor \cdot \rfloor$, 21
- $\vdash ::$, 22
- $\vdash \llbracket \cdot \rrbracket$, 22
- $\vdash \leq$, 25
- $\vdash (\leq)_w$, 25
- $\vdash (\leq)$, 25
- $\vdash \sqrt{\cdot}$, 25
- $\vdash \llbracket \cdot \rrbracket$, 26
- $\vdash ::$, 26
- $\vdash : \sqrt{\cdot}$, 27
- $\xrightarrow{\textit{jvm}}_1$, 31
- $\xrightarrow{\textit{jvm}}$, 31
- $\xrightarrow{\textit{djvm}}_1$, 34
- $\xrightarrow{\textit{djvm}}$, 34
- \sqsubseteq_r , 38
- \sqcup , 38
- $\lfloor \sqsubseteq_r \rfloor$, 40
- $\{ \leq_r \}$, 42
- $\vdash , ::$, 51
- $\vdash_1 \langle \cdot \rangle \Rightarrow \langle \cdot \rangle$, 57
- $\vdash_1 ::$, 58
- \subseteq_m , 61
- $\triangleright /$, 64
- \triangleright , 64
- \triangleright , 64
- $\vdash , \llbracket \cdot \rrbracket$, 68
- \mathcal{A} , 23
- \textit{acc} , 38
- \textit{Addr} , 4
- \textit{addr} , 4, 5
- $\textit{addr-of-sys-xcpt}$, 11
- \textit{app} , 42, 50
- \textit{app}_i , 47
- $\textit{arbitrary}$, 33
- $\textit{assigned}$, 14
- \mathcal{B} , 57
- \textit{bcv} , 43
- \textit{binop} , 5, 38
- \textit{blank} , 31
- \textit{blocks} , 15
- \textit{Bool} , 4
- \textit{bool} , 2
- $\textit{Boolean}$, 20
- $\textit{bounded}$, 42
- C' , 46
- \textit{Cast} , 5

cdecl, 6
check, 33
check-instr, 33, 34
Checkcast, 30
Class, 20
class, 6
ClassCast, 10
closed, 38
CmpEq, 30
cname, 4
coalesce, 41
compC, 60
compE₁, 59
compE₂, 60
compM, 60
compP, 60
compP₁, 61
compP₂, 61
compT, 68
compT_a, 67
compxE₂, 61

D, 23
distinct, 3
distinct-fst, 17
dom, 3
drop, 33

ebinop, 39
eff, 42, 50
eff_i, 48
empty, 3
Err, 39
err, 39
err-semilattice, 39
Err.esl, 39
Err.le, 39
Err.sup, 39
error, 43
esl, 45
ex-table, 29
exception, 64
exec, 30
exec_d, 33
exec-instr, 32
exp, 56
expr, 4
expr₁, 56

false, 5
fdecl, 6
field, 30
fields, 8
filter, 3
final, 12
finals, 13
find-handler, 30, 31
frame, 29
fst, 2
fv, 19
fvs, 19

Getfield, 30
Goto, 30

hd, 31
heap, 8
hidden, 63
hp, 8

IAdd, 30
if else, 5
IfFalse, 30
init-fields, 8
instrs-of, 30
int, 47
int (type), 2
Integer, 20
Intg, 4
Invoke, 30
is-class, 6
is-Intg, 35
is-refT, 20
is-type, 20

J-mb, 6
J-prog, 6
J₁-prog, 56
J2JVM, 61
jvm-method, 29
jvm-prog, 29
jvm-state, 29

kildall, 44

lcl, 8
le, 39
lift2, 39
list, 3, 40
Listn.le, 40
Listn.sl, 40
Listn.sup, 41
Load, 30
locals, 8
lub, 45

map, 3
map-of, 3
map-snd, 43
map2, 40
max-stack, 62
max-var, 58
mdecl, 6
method, 30
mname, 4
mono, 42
mxl, 46, 67
mxl₀, 29
mxs, 29, 46, 67

nat, 47
nat (type), 2
New, 30
new, 5
new-Addr, 31
None, 3
norm-eff, 48
Normal, 33
NT, 20
Null, 4
null, 5
NullPointer, 10

obj, 8
OK, 39
ok-val, 48
opstack, 29
opt, 40
Opt.esl, 40
Opt.le, 40
Opt.sup, 40
option, 3

ord, 38
order, 38
OutOfMemory, 10

P, 46, 67
pc, 41
pcs, 64
Pop, 30
preallocated, 25
preserves, 42
Product.esl, 40
Product.le, 40
Product.sup, 40
prog, 6
Push, 30
Putfield, 30

reg-sl, 46
registers, 29
relevant-entries, 49
replicate, 33
Return, 30
rev, 31

semilat, 38
set, 3
shift, 69
sl, 46
sl (type), 38
snd, 2
SOME, 44
Some, 3
stable, 42
start-heap, 35
state, 8
states, 50
step, 41, 42
stk-esl, 46
Store, 30
succs, 48
sup, 39
sys-xcpts, 10

take, 31
the, 3
the-Addr, 31
this, 29
Throw, 30

throw, 5
tl, 31
 T_r , 46, 67
true, 5
try catch, 5
 T_s , 46
ty, 20
 ty_i , 45
 ty_i' , 45
 ty_l , 45
 ty_P , 51
type-error, 33
TypeError, 33
typeof, 20, 21
types, 45
 ty_s , 45

Unit, 4
unit, 5
unmod, 63
upto-esl, 41

Val, 5
Var, 5
vname, 4
Void, 20

wf-cdecl, 17
wf-fdecl, 17
wf-J-mdecl, 24
wf-J₁-mdecl, 58
wf-J-prog, 24
wf-J₁-prog, 58
wf-jvm-prog, 51
wf-jvm-prog_k, 51
wf-mdecl, 17
wf-mdecl-test, 17
wf-prog, 17
wf-syscls, 17
while, 44
while, 5
wt-app-eff, 43, 50
wt-kildall, 51
wt-method, 50
wt-start, 50
wt-step, 42
wwf-J-mdecl, 18

wwf-J-prog, 18

xcpt-app, 49
xcpt-eff, 50
xt, 46