A Coq Formalization of a Type Checker for Object Initialization in the Java Virtual Machine

Yves Bertot

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Coq formalization of a Type Checker for Object Initialization in the Java Virtual Machine

Yves Bertot

N° 4047

Novembre 2000

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4047--FR+ENG

A Coq formalization of a Type Checker for Object Initialization in the Java Virtual Machine

Yves Bertot

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lemme

Rapport de recherche n° 4047 — Novembre 2000 — 53 pages

Abstract: We worked on a type system proposed in [11] to enforce a discipline for object initialization in the Java Virtual Machine language, to show how this type system could be implemented in the Coq proof and specification language. We used this description both to prove the theorems of [11] and to construct an effective verifier for this discipline.

Key-words: programming language semantics, Java, byte-code, Coq, automatic demonstration

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Une formalisation en Coq d'un vérificateur de types pour l'initialisation des objets dans la Machine Virtuelle Java

Résumé : Nous avons travaillé sur un système de types proposé dans [11] pour imposer une discipline d'initialisation des objets dans le langage de la machine virtuelle Java. Nous avons montré que ce système de type pouvait être décrit à l'aide de Coq. Nous avons utilisé cette description pour démontrer mécaniquement les théoremes de [11] et pour construire un vérificateur effectif pour cette discipline de types.

Mots-clés : sémantique des langages de programmation, Java, byte-code, Coq, Démonstration automatique

1 Introduction

We worked on a type system proposed in [11] to enforce a discipline for object initialization in the Java Virtual Machine language, to show how this type system could be encoded into the Coq proof and specification language. We used this description both to prove the theorems of [11] and to construct an effective verifier for this discipline.

Thanks to the extraction mechanism provided in Coq [22], we obtain a program in CAML that can be directly executed on sample programs. In the future, this will make it possible to study the complexity of the algorithm and to propose enhancements.

1.1 Related work

The byte-code verifier is advertised as a key component of the security and safety strategy making it possible to use and exchange Java programs without fearing too much damage due to erroneous programs or even malignant program providers. As Java is likely to become one of the languages used to embed programs in all kinds of appliances or computer-based applications, it becomes important to verify that the claim of safety is justified.

Several teams around the world have been working on verifying formally that the properties of the Java language and its implementation suite make it a reasonably safe language. Some of the work done is based on pen-and-paper proofs that the principles of the language are correct, see for instance [27, 12, 19].

Closer to our concerns are the teams that use mechanical tools to verify the properties established about the formal descriptions of the language. A very active team in this field is the Bali team at University of Munich who is working on a comprehensive study of the Java language, its properties and its implementation [20, 18, 24] using the proof system Isabelle [23]. Other work has been done with the formal method B and the associated tools [3] or at Kestrel Institute using Specware [13, 25].

1.2 A few facts about Coq

Coq is a proof system based on type theory [16], and more precisely on the calculus of constructions [4, 5] and its inductive extensions [6, 21].

An interesting feature about type theory based proof systems like Coq is that they contain a typed purely functional programming language with a notion of types that is powerful enough to express almost everything one wants to specify about programs[1]. A second feature is the uniform treatment of data-types and logical proposition, following the discipline known as the *Curry-Howard* principle.

Because types are used to denote both regular data-types and propositions, functions may take as arguments plain data and proofs that the data verifies some property. Similarly, functions may return data accompanied with proofs that the returned data satisfies some other property. This makes it possible to simulate the kind of reasoning about programs

[1] one notable exception is information about algorithmic complexity

that is usual with axiomatic semantics [10, 14]. Extensions also handle imperative programs [9, 8].

2 Formalizing the language and type system

2.1 Testing for equality and dependent types

In this paper, we will consider objects inhabiting several types of data. For instance, there will be a type for classes, a type for values, etc. For most of these types, we will suppose the existence of a function that tests whether two terms are equal.

In regular programming, such a function would be represented using a boolean function, returning `true` when the two compared values are equal and `false` when they are different. Making sure the `true` value is returned when it should is the programmer's responsibility.

Using dependent types, it is possible to specify equality functions in a more precise way. In fact, we will use functions that return *proofs*. When the two terms are equal, the returned value will be a proof that they are equal. When the two terms are distinct, the returned value will be a proof that they are not equal. When `t` and `t'` are two terms in the same type, the type of proofs that `t` and `t'` are equal or proofs that `t` and `t'` are not equal can be represented as follows, where `~` represents negation.

```
{t=t'}+{~ t=t'}
```

Obviously, the type returned by this function depends on the values `t` and `t'`. This is an occurrence of *dependent types*. The type of the equality function for an arbitrary type `A` is then represented as follows:

```
(t,t':A){t=t'}+{~t=t'}
```

By contrast with usual (non-dependent functions), we see that the notation for this type uses the names of the two arguments, `t` and `t'`, to describe the result type.

There may be several implementations of the equality test function, with various degrees of efficiency, but we see that the type specifies exactly what its returned value must be. This example also helps showing that dependent types can be strong enough to give a specification of some code behavior.

The value returned by an equality test function can be used in a `Case` construct. If `eq_A_dec` is an equality test function, then a `Case` construct can have the following form:

```
Cases (eq_A_dec t1 t2) of
   (left h) => B1
  |(right h') => B2
end
```

In this expression, it is possible that `h` occurs in `B1` or `h'` occurs in `B2`. One describes in `B1` what computation should take place when we have a proof `h` that `t1` and `t2` are equal and

in `B2` what computation should take place when we have a proof `h'` that `t1` and `t2` are not equal.

In Coq, it is possible to define functions whose value ranges over types. This corresponds to the well-known polymorphism of functional languages like ML [17]. We use this to describe in a general manner the type of equality test functions associated to an arbitrary data type:

```
Definition eq_fun : Set -> Set :=  [A:Set](x,y:A){x=y}+{~ x=y}.
```

As far as notations are concerned, depend types may sometimes be written using a capital Π, when the returned value is in some data-type, or using a universal quantification symbol $\forall$, when the returned value is the proof of some proposition. In this paper, we will often use the notation with parentheses that corresponds to the exact text sent to the Coq proof assistant. In mathematical notations, we will sometimes use the $\forall$ notation, but rarely the Π notation. Using mathematical notations, one writes the following definition for `eq_fun`:

$$\texttt{eq\_fun} = \lambda A : Set. \forall x, y : A. \{x = y\} + \{\neg x = y\}$$

2.2 Representing object types

The programs studied in this experiment are constructed with a very restricted subset of the instructions available in the Java Virtual Machine language. We will reason about objects belonging to various classes, but the classes really won't matter much. For this reason, we only assume there exists some type `T` whose inhabitants are the classes, and for programming purposes, we assume that there exists an equality test function for this type. These assumptions are written down with the following command.

```
Parameters T:Set; eq_type_dec:(eq_fun T).
```

In our work, we will also need to refer to addresses in the program and to variables. Instead of choosing concrete data-types to represent these kinds of data, we prefer to let them unspecified. Still, we need to know that some functions exist to compare values in these data types. The type `ADDR` will be used to denote addresses in programs, while the type `VAR` will be used to denote variable locations, and the functions `eq_addr_dec` and `eq_var_dec` will be used to compare values in these types.

```
Parameters ADDR, VAR:Set; eq_addr_dec : (eq_fun ADDR);
   eq_var_dec: (eq_fun VAR).
```

The values manipulated in instructions are either initialized or uninitialized objects in arbitrary classes or integers. Since we are studying a type-checker, we will have to use a formal language to describe the possible types of values. In proof systems, the usual practice is to represent formal languages as inductive data-types. It is what we do with this formal language of types, even though the language is not really inductive, in the sense that a type description may not be a sub-component of another.

```
Inductive jtype: Set :=
  int : jtype
| obj_type: T -> jtype
| un_type: T -> ADDR -> jtype
| top: jtype.
```

We must make two remarks about this description. First, uninitialized objects belonging to the same class but created at two different places in the program are not in the same type. This reflects that the discipline given in [11] imposes that an object created at some location in the program should be initialized before a new object can be created at the same location. To enforce this constraint, but allow two objects of the same class to be uninitialized at the same time when they have been created at different location, the type system considers that the location is part of the object type. The second remark concerns the introduction of a type called `top`. This type does not really represent an effective object class, but it reflects that a program can be well-typed even if the type of some variables at some line is unknown.

2.3 Representing the object language

The studied language contains a very abstract subset of the commands present in the JVM language. Basically, it contains an example of each basic instruction, plus instructions related to object creation, initialization and use. Representing such a language is best done with and inductive definition:

```
Inductive jvmli: Set :=
     inc: jvmli
    | pop: jvmli
    | push0: jvmli
    | load: VAR ->jvmli
    | store: VAR ->jvmli
    | if0: ADDR ->jvmli
    | new: T ->jvmli
    | init: T ->jvmli
    | use: T ->jvmli
    | halt: jvmli .
```

Automatically, Coq's processing of this inductive definition yields a pattern-matching construct that makes it possible to define functions or to reason by cases. This construct is used automatically by some tactics to support case reasoning. For instance, suppose we want to prove some fact about an arbitrary instruction, of the following form

$$C[inst]$$

where *inst* is a variable of type `jvmli`. If we run the command `Elim inst`, then the system will make it possible to prove 10 goals, where *inst* has been replaced by the various possible instructions:

$$C[\texttt{inc}] \quad C[\texttt{pop}] \quad C[\texttt{push0}] \quad \forall v : VAR.\ C[(\texttt{load}\ v)] \quad \ldots$$

2.4 Using dependent types

In [11], the authors propose to partition the set of possible values into subsets corresponding to each possible type. Using dependent types, it is possible to represent all these types with a family of sets, indexed by types. We do this by considering there exists a function from the set of possible class types to sets. Similarly, the uninitialized values must be distinguished from the plain values, with an extra index to also distinguish values initialized at different locations.

```
Parameter object_value:T -> Set.
Parameter uninitialized_value : T -> ADDR -> Set.
```

Using this approach, a value is either an integer, a plain object, or an uninitialized object. To describe this, we use the following definition of the type `value`:

```
Inductive value: Set :=
     int_val: integer ->value
    | obj: (t:T) (object_value t) ->value
    | un: (t:T) (a:ADDR) (uninitialized_value t a) ->value.
```

As we see, the last two constructors of this "inductive" type are dependent: the type of the second constructor's second argument depends on the value of the first argument, for instance.

Later in our work, we have had reasons to question this choice of encoding. One usual property of constructors is that they are injective, so that one can easily construct a collection of theorems expressing that if two expressions built with the same constructor are equal, then the constructor's arguments must be pairwise equal. Here in the case of `obj`, this would yield the following two theorems:

$$\forall t_1, t_2 : T. \forall o_1 : (object\_value\ t_1). \forall o_2 : (object\_value\ t_2).\\ (obj\ t_1\ o_1) = (obj\ t_2\ o_2) \Rightarrow t_1 = t_2$$

$$\forall t_1, t_2 : T. \forall o_1 : (object\_value\ t_1). \forall o_2 : (object\_value\ t_2).\\ (obj\ t_1\ o_1) = (obj\ t_2\ o_2) \Rightarrow o_1 = o_2$$

However, the second theorem cannot be constructed: its statement is not well-typed. Since o_1 has type $(object\_value\ t_1)$ and o_2 has type $(object\_value\ t_2)$, these types are not convertible and $t_1 = t_2$ is not well-typed, because the usual equality relation in Coq is well-typed only when the two objects are in the same type. Actually, equality is a *three-place* predicate, where an invisible first argument is the type of the other two.

A solution to this problem is to use a *dependent* equality. This kind of equality is different from the usual equality in Coq in that it has four arguments: two arguments are the objects being compared, while two other arguments are their types (which may then be different). This dependent equality makes is possible to write down expressions about the equality of two terms even though their types maybe different.

In practice, a few standard tools provided in Coq to handle inductive types, in particular tactics `Injection` and `Inversion` will behave in an unexpected manner as soon as one has

such dependent constructors, because these tactics typically work with the usual 3-place equality.

2.5 Operational semantics

The operational semantics describes the behavior of each instruction as a transformation of some machine state. The machine state combines values associated to each local variables, represented by a total function from `VAR` to the type of values, a local stack, represented by a list of values, a program counter and a program. The program has a special status since it is never modified during execution, and we actually do not include it in our formalization of the machine state. We name the type of possible machine states `istate`.

The type `istate` is defined with a constructor `mkistate` with the following type:

$$\texttt{mkistate} : \texttt{ADDR} \rightarrow \texttt{frame} \rightarrow \texttt{stack} \rightarrow \texttt{istate}$$

In mathematical notations, we will write $\langle pc, f, s\rangle$ for `(mkistate pc f s)`. The type `frame` itself is an abbreviation for the type `VAR` $\rightarrow$ `value`, and the type `stack` is an abbreviation for `(list value)` (the type of lists of values).

Each of the arguments of the constructor correspond to a field in a record. The first field, of type `ADDR`, represents the *program counter*, indicating where in the program execution is currently taking place. The second field, of type `frame`, describes the values of all variables in the program, the third argument, of type `(list value)` represents the stack.

Execution is represented by a relation, named `istep`, between three pieces of data. The first is the program, a function from addresses to instructions, the second is the machine state before executing, of type `istate`, the third is the machine state after executing, also of type `istate`.

$$\texttt{istep} : \texttt{program} \rightarrow \texttt{istate} \rightarrow \texttt{istate} \rightarrow \texttt{Prop}$$

When reading the predicate (`istep` P s_1 s_2), one can verbalize this as the sentence s_2 *is the result of one step execution starting from* s_1 *when the executed program is* P. In this sense, P and s_1 are input and s_2 is output.

It may seem more practical to describe execution simply as a function from pairs of program and machine state to machine states. However, this would impose that execution is total and deterministic.

Total execution would mean that all instructions can be executed in any possible machine state. This is inadequate, since we actually want to express properties of programs in terms of execution failure. This could be recovered by adding a dummy state to represent failed execution.

Deterministic execution has practical advantages: knowing that there can be only one output for each possible input can make writing down various properties much easier. However, it can also turn out to be an overkill. For abstraction purposes, it may be easier to describe some parts of the process as non-deterministic, to avoid a need to be too precise about the actual operation of the machine, the same way as non-deterministic choice is used in other formal methods to describe under-specification.

The relation `istep` is best defined as an inductive proposition. In such a definition, one provides a collection of theorems, also called *constructors*, that express when the proposition is provable. The relation that is defined is the smallest one that respects these theorems, and constraints on the form of definitions ensure that such a relation exists. Coq's processing of the inductive definition generates a `Case` construct, which expresses that propositions of the form `(istep P t1 t2)` can only have been proved using one of the constructors. As for the definition of the language `jvmli` above, the definition is not really inductive, but it will be handy to be able to reason by cases about this proposition.

```
Inductive istep[P:program]: istate -> istate ->Prop :=
  js_inc:
   (pc:ADDR) (f:frame) (n:integer) (s:stack) (P pc)=inc ->
   (istep
      P (mkistate pc f (cons (int_val n) s))
       (mkistate (add1 pc) f (cons (int_val (increment_int n)) s)))
| js_pop:
    (pc:ADDR) (f:frame) (v:value) (s:stack) (P pc)=pop ->
    (istep P (mkistate pc f (cons v s)) (mkistate (add1 pc) f s))
...
```

All constructors have a similar shape, that can be summarized as follows:

$$\forall pc, x_1, \ldots x_k.(P\ pc) = inst \Rightarrow \textit{side conditions} \Rightarrow (\texttt{istep}\ P\ \langle pc, \ldots\rangle\ \langle \ldots\rangle)$$

They describe the behavior for one instruction, as expressed by the fact that $(P\ pc)$ refers to that instruction. All instructions are represented, except one, `halt`. This instruction can never be executed.

The side conditions given among the premises of each constructor restrict the conditions under which the instruction may be executed. Some other conditions are expressed by the pattern that the input state, $\langle pc, \ldots\rangle$ should match. For instance, the pattern for the input state in the constructor `js_inc` above expresses that `inc` will operate normally only if the stack is not empty and the value on top of the stack is an integer value.

2.6 Object creation, initialization and use

The object of this work is to study the way objects are created, initialized and used. This appears in the operational semantics, in the treatment of the three instructions `new`, `init`, and `use`. The constructor of `istep` for `new` has the following form:

```
js_new:
  (pc:ADDR; f:frame; t:T; s:stack; a:(uninitialized_value t pc))
  (P pc)=(new t) -> (u_unused t pc a f s) ->
  (istep P (mkistate pc f s)
       (mkistate (add1 pc) f (cons (un t pc a) s)))
```

There is one side condition, requiring that there is no uninitialized object in the whole memory of the form `(un pc a)`. This side condition expresses that any operational semantics for the `new` instruction is acceptable, as long as it actually creates a new object, that is not already present in the memory. In this sentence, the memory actually is the combination of `f`, the values of all variables, and `s` the stack.

The constructor of `istep` for `init` has the following form:

```
| js_init:
  (pc, pc':ADDR; f:frame; t:T; s:stack; a, a': value;
   a0:(uninitialized_value t pc'); a'0:(object_value t))
  (P pc)=(init t) ->
  a=(un t pc' a0) -> a'=(obj t a'0) ->
  (o_unused t a'0 f s) ->
  (istep P (mkistate pc f (cons a s))
           (mkistate
              (add1 pc) (subst_v f a  a')
              (subst_stk_v s a a')))
```

The salient feature in this constructor is the way all instances of the uninitialized object in the memory are substituted with the initialized object. Substitution relies on the assumption that there exists two generic functions, one for substituting in maps from the type `VAR` to values `subst_v` and one for substituting in stacks of values `subst_stk_v`. these two functions actually are instances of more general functions that are defined for any type provided an equality test function is provided for that type. For instance, `subst_stk_v` is defined as follows:

```
Definition subst_stk_v :
   (list value) -> value -> value -> value : :=
  (subst_stk value eq_value_dec).
```

The constructor of `istep` for `use` has the following form:

```
| js_use:
  (pc:ADDR; f:frame; t:T; a:(object_value t); s:(list value))
  (P pc)=(use t) ->
  (istep P (mkistate pc f (cons (obj t a) s))
           (mkistate (add1 pc) f s)) .
```

The authors of [11] have chosen an excessively abstract representation of the notion of using an object: basically, the `use` instruction specifies the class of the object on which to operate. It is only necessary to verify that the object belong to the specified class and the operation does not modify the object. This is really far from actual operation in the real-life language.

2.7 Static semantics

The structure of the static semantics is very similar to that of the operational semantics. We almost find the same rules, with a structure that is very close. The main differences appear in the form of the typing relation, the collapse of two behavior rules into only one for the control instruction `if`, and the presence of a rule for the `halt` instruction.

The typing relation has the form `(iwtstep P F S bound i)`, where the arguments are described as follows:

- `P` is a program, that is, a total function from addresses to instructions;
- `F` is a total function from addresses to total functions mapping variables to types, used to represent static information about the type of variables at each line of the program;
- `S` is a total function from addresses to lists of types, used to represent static information about the stack at each line of the program;
- `bound` is an address, used to represent the last valid address in the program;
- `i` is an address, used to represent the address in the program currently under scrutiny.

The relation `(iwtstep P F S bound i)` can be read as: *the program* `P` *of length* `bound` *is consistent at address* `i` *with the type information given by* `F` *and* `S`. This relation is defined as an inductive proposition.

```
Inductive iwtstep[P:program; F:ADDR ->(map jtype);
                  S:ADDR ->(list jtype); bound, i:ADDR]: Prop :=
 wt_inc:
 (alpha:(list jtype))
 (P i)=inc -> (same_map jtype (F (add1 i)) (F i)) ->
 (S (add1 i))=(S i) ->
 (S i)=(cons int alpha) -> (smaller (add1 i) bound) ->
 (iwtstep P F S bound i)
| wt_pop:
 (tau:jtype)
 (P i)=pop -> (same_map jtype (F (add1 i)) (F i)) ->
 (S i)=(cons tau (S (add1 i))) -> (smaller (add1 i) bound) ->
 (iwtstep P F S bound i)
...
```

The proposition `(same_map jtype (F (add1 i)) (F i))` expresses that the two functions `(F (add1 i))` and `(F i)` are point-wise equal. When we use mathematical notations rather than Coq notation, we will write $F_{i+1} = F_i$ for the same concept, even though Coq does not consider two functions that are point-wise equal to be equal.

At this point, there is a strong technical difference between the formalization described in [11] and ours. Freund and Mitchell explicitly use the fact that `P` is a *partial* function in

their work, and the various inference rules for the type system they propose rely on premises of the form "$P(i)$ *is defined*". In our work, it is much more practical to describe `P` as a total function. In this case, we have to replace the property of P being defined for some address by some other property of this address. We have chosen to consider that the address has to be smaller than some bound, with the intuitive objective that the bound represents the address of the last instruction in the program.

We have left the set of addresses as an unknown set, but we have assumed it comes with two specific elements, `zero_addr` and `bound`, an increment function `add1` and an order `smaller` that has intuitively the same properties as the order "less-or-equal" on natural numbers, except that the only numbers related by this order are those numbers that are smaller than `bound`.

In the relation `(iwtstep P F S bound i)`, `F` and `S` are assumed to be some known input, describing static information about the types in the program. The constructors of this relation describe what it means for the program to be consistent with this information, but they do not describe explicitly how this information was constructed in the first place.

2.8 well-typed programs and reachability

The authors of [11] lift the property of being well-typed to complete programs by stating that all lines have to be well-typed. We use a weaker notion of well-typed programs where only the reachable instructions have to be well-typed. The reasons for this weakening is that we will prove that the weak notion suffices. Moreover, our work contains a second part: providing an algorithm that shows a program to be well-typed. Having a weaker property to establish will make this algorithm easier to write.

We define reachability as the transitive reflexive closure of a one step reachability relation that expresses that address j is one-step reachable from i if the instruction at address i in the program is not a `halt` instruction and $j = i + 1$ or if the instruction at address i in the program is `if0` j. The relation of one-step reachability is called `cg1_reachable`, while the reflexive transitive closure is called `cg_reachable`. Both relations are defined inductively.

We prove that this notion of reachability is relevant by relating it with execution, in the theorem that follows:

`istep_reachable`:
$\forall P, pc, f, s, pc', f', s'.$(`istep` $\langle pc, f, s, \rangle$ $\langle pc', f', s' \rangle$) $\Rightarrow$ (`cg1_reachable` pc pc') This theorem is easily proved by a systematic analysis of all the constructors of `istep`.

2.9 Static semantics for creation and use

The constructor of `iwtstep` for the instruction `new` has the following form:

```
wt_new:
    (sigma:T) (P i)=(new sigma) ->
    (same_map jtype (F (add1 i)) (F i)) ->
    (S (add1 i))=(cons (un_type sigma i) (S i)) ->
```

```
    (not_in jtype (un_type sigma i) (S i)) ->
    ((y:VAR)~ (F i y)=(un_type sigma i)) ->
    (smaller (add1 i) bound) ->
    (iwtstep P F S bound i)
```

The premise `(same_map ...` expresses that the variables after executing the instruction have the same type, the premise `(S (add 1 i))= ...` expresses that the type of the stack after executing the instruction is almost the same as the type of the stack before executing, except that an uninitialized object has been added on top. At this place, it is also necessary to check that no other object of the same type already exists in memory, with the premise `(not_in ...`, and the universally quantified premise `((y:VAR)...`. The last thing that needs checking, is that the next address is also valid, since the `new` instruction transfers the control to the following instruction. This is done with the premise `(smaller (add1 i) bound)`.

Obviously, the most important premises of this constructor are the fourth and fifth, that enforce the constraint about having no instance of an uninitialized object left from a previous object creation at the same address.

The constructor of `iwtstep` for the instruction `init` has the following form:

```
wt_init:
    (sigma:T) (j:ADDR) (alpha:(list jtype))
    (P i)=(init sigma) ->
    (same_map
       jtype (F (add1 i))
       (subst_t (F i) (un_type sigma j) (obj_type sigma))) ->
    (S i)=(cons (un_type sigma j) alpha) ->
    (S (add1 i))= (subst_stk_t alpha
                        (un_type sigma j) (obj_type sigma)) ->
        (smaller (add1 i) bound) ->(iwtstep P F S bound i)
```

The salient feature of this constructor is that it does not operate only on the top of the stack, but that modifications are also performed on the rest of the stack and the variables. This expresses that initialization is visible through all references to the object. Consistency of this constructor with the corresponding constructor in the operational semantics is made easier by the fact that we use the same `subst` and `subst_stk` functions to perform this pervasive update.

2.10 Relating dynamic and static information

The static information given in F and S is an abstraction of the information that should be given in the virtual machine state for any execution and any line of the program. We need to have a way to express that the information in F and S for some address pc indeed is a valid abstraction of some machine state.

Everything relies on a relation `type_value` that associates a type with any value, and a theorem `type_value_det`, that says a value can only have one type. The function is easy to write, knowing that every object actually is a record associating a type tag and a value in the corresponding type. Proving the theorem is also straightforward.

The first part is that all variables in the state must be given values (through `f`) that have the type ascribed to them in `F`. We express this with a relation `type_map`, such that (`type_map` f $(F\ pc)$) is equivalent to the proposition:

$$\forall x : VAR.(\texttt{type\_value}\ (f\ x)(F\ pc\ x)).$$

The second part is that all values on the stack have the type ascribed to them in S. This part uses a new predicate `type_stack` that lifts `type_value` to stacks in a natural way.

2.11 Initialization consistency

To maintain the relation between dynamic and static information, some extra consistency must be verified. For initialization, substitutions are used on both the dynamic and the static side, but we need to make sure that the locations modified by this substitution are the same on both side. Type correspondence is not enough to ensure this here, because several object can have the same type. The property $ConsistentInit$ introduced in [11] takes care of the extra requirements. We implement it in Coq with a property that has the same name, but we use a different typography to indicate that it is part of our formal development.

The property $ConsistentInit$ ensures that all locations that have the same uninitialized type also carry the same uninitialized value. In the definition of `jvmli` types, uninitialized types have the form (`un_type` $\tau\ i$), where τ is a class and i is an address. The characterization for the type map F the value map f, a class τ, an address i, and an uninitialized value b is expressed with the following formula:

$$\forall x : VAR.(F\ x) = (\texttt{un\_type}\ \tau\ i) \Rightarrow (f\ x) = (\texttt{un}\ \tau\ i\ b)$$

Characterizing the same kind of property for all locations on stacks is described with an inductive definition called `StackCorresponds`. We will write

$$(\texttt{StackCorresponds}\ t\ i\ b\ S\ s)$$

to express that if a location in `S` has type (`un_type` $t\ i$), then the corresponding location in `s` contains the value (`un` $t\ i\ b$). This relation also expresses that S and s have the same height. Expressing correspondence between the uninitialized type (`un_type` $t\ i$) and the uninitialized value (`un` $t\ i\ b$) for all memory locations, variables and stack cells alike, is expressed with a relation of the form

$$(\texttt{corresponds}\ t\ i\ b\ F\ S\ f\ s)$$

This relation is defined as an inductive proposition with the following formal specification:

```
Inductive corresponds
  [t:T; i:ADDR; b:(uninitialized_value t i); F:(map jtype);
   S:(list jtype); f:(map value); s:(list value)]: Prop :=
 corr:((x:VAR) (F x)=(un_type t i) ->(f x)=(un t i b)) ->
      (StackCorresponds t i b S s) ->(corresponds t i b F S f s).
```

Full consistency with respect to initialization requires that there exist a value for every uninitialized type characterized by the pair of an address and a class type, as expressed by the following formal definition of `ConsistentInit`:

```
Inductive ConsistentInit [F:(map jtype);S:(list jtype);
                          f:(map value);s:(list value)]: Prop :=
 cons_init:
  ((t:T) (i:ADDR)(Ex [b:(uninitialized_value t i)]
                      (corresponds t i b F S f s))) ->
 (ConsistentInit F S f s).
```

3 First consistency proof

The one-step soundness theorem of [11] simply expresses that type-correspondence and initialization consistency are preserved throughout execution for well-typed programs.

The main structure of the proof for this theorem is described in figure 1 (the graph was obtained mechanically, using the graph tools for Coq described in [2]), where some auxiliary lemmas have been removed for the sake of readability. This graph shows that the proof of soundness is broken down in a proof for each possible instruction (fifth column in the graph layout). Also a lot of the complexity resides in lemmas around the relation *ConsistentInit*. In this section, we will only study some parts of the proof. First, we are going to look at proofs around typing, then we are going to study several aspects of the proofs around the relation *ConsistentInit*.

3.1 Relating typing and operational semantics

According to the operational semantics (relation `istep`), each step of execution produces a new execution state by a simple modification of the previous one. Execution states are composed of a mapping of variables to values and a stack, and modifying states means updating a variable, modifying the value on top of the stack, or substituting all occurrences of a value with a new value. Similarly, for each line of the program, the typing specification (relation `iwtstep`) establishes that the typing information at two different lines must be related in some way: for instance the second line must be the result of updating the type for a variable, modifying types on top of the stack, or substituting all occurrences of a type with a new type. For the `if0` instruction, type information for three lines is compared.

When a program is well-typed, we need to show that the correspondence between type information and operational data is preserved through execution.

Figure 1: Data-types and theorems for the one-step soundness theorem

For instance, the semantics for the `store` instruction requires that the mapping from variables to values is updated to associate a new value to a given variable. This operation is represented by the function `update`. Fortunately, the same update operation (but for different types) is used in the type system. The theorem that is needed to show the link between both operations has the following form:

`update_type`
$(F' \simeq (\texttt{update}\ jtype\ F\ y\ t) \wedge f' \simeq (\texttt{update}\ value\ f\ y\ v) \wedge$
$\quad (\texttt{type\_value}\ (f'\ x)\ (F'\ y)) \wedge (\texttt{type\_map}\ f\ F)) \Rightarrow (\texttt{type\_map}\ f'\ F')$

We use $\simeq$ to express that two functions are point-wise equal. In Coq, this is usually not enough to express that the functions are the same object. This is expressed by an hypothesis called `same_map_def`.

`same_map_def`:
$(\texttt{same\_map}\ f\ g) \Rightarrow \forall x.(f\ x) = (g\ x)$

The proof in our formal development has the following form:

```
Theorem update_type:
  (same_map jtype F' (update ? F y t)) ->
  (same_map value f' (update ? f y v)) ->
  (type_value v t) -> (type_map f F) ->(type_map f' F').
Unfold type_map; Intros H' H'0 H'1 H'2 x.
Rewrite same_map_def with 1 := H'0;
Rewrite same_map_def with 1 := H'.
Case (eq_var_dec x y).
Intros H'3; Rewrite H'3; Do 2 Rewrite update_spec1; Auto.
Intros; Repeat Rewrite update_spec2; Auto.
Qed.
```

The theorems `update_spec1` and `update_spec2` used here express that for any type A, variable x and y, and values v_1 and v_2 in A, (`update` $A\ f\ x\ v_1\ v_2\ y$) is equal to v_1 or v_2 depending on whether x is equal to y or not, respectively.

The `store` and `init` instruction actually are the only instructions where modifications of variable values occur. For all the other instructions, verifying that the variables keep values that have a type compatible with the type imposed by the static semantics is a trivial matter, because nothing changes, as expressed by the following theorem:

`same_type_map`:
$\quad \forall f, g, F, G.f \simeq g \Rightarrow F \simeq G \Rightarrow (\texttt{type\_map}\ f\ F) \Rightarrow (\texttt{type\_map}\ g\ G)$

Here again,the proof is quite trivial:

```
Theorem same_type_map:
  (f, g:(map value); F, G:(map jtype))
  (same_map ? f g) -> (same_map ? F G) -> (type_map f F) ->
```

```
  (type_map g G).
Unfold type_map same_map.
Intros f g F G H' H'0 H'1 x; AutoRewrite [[H' RL H'0 RL]]; Auto.
Qed.
```

3.2 Relating `ConsistentInit` and simple state modifications

Many lemmas express that the *ConsistentInit* property is preserved through memory updating, substitution, and stack operations. Many of these lemmas are usually proved by showing that *ConsistentInit* and the update operations follow the same inductive structures.

For instance, we can look at the proof for property `not_in_to_StackCorresponds`, which expresses that stacks are coherent with the initialization discipline for a type and an address as long as the type of uninitialized objects created at that address does not appear in the stack:

(`type_stack` s S) $\Rightarrow$ (`not_in` (`un_type` t pc) S) $\Rightarrow$
(`StackCorresponds` t pc a S s).

The text for this statement and proof appears as follows in our formal development.

```
Theorem not_in_to_StackCorresponds:
  (t:T; pc:ADDR; a:(uninitialized_value t pc); S:(list jtype))
  (s:(list value)) (type_stack s S) ->
  (H_not_in:(not_in (un_type t pc) S))
  (StackCorresponds t pc a S s).
Proof.
Intros t pc a S s Htype_stack; Elim Htype_stack; Intros;
 Inversion H_not_in; Auto.
Qed.
```

Please notice that the assumption `(not_in t pc)` is named in this theorem statement. Although this makes the statement less readable, this property is used by the proof commands. In the proof, `H_type_stack` is the hypothesis `(type_stack s S)` and the command `Elim H_type_stack` instructs the proof engine to perform a proof by induction on the structure of the derivation for this proposition. This generates two cases, because `type_stack` is an inductive proposition with two constructors. The proof command

```
Intros; Inversion H_not_in; Auto
```

is applied on both goals. In each case, the hypotheses have been generated so that the hypotheses derived from the `not_in` premise of the theorem are named `H_not_in`. The proof proceeds by a case analysis of the way the hypothesis `H_not_in` could have been proved. This case analysis, when done with the `Inversion` tactic, discards the cases that are incompatible

with the observed values. For the cases that are not discarded by `Inversion`, automatic proof search using the `Auto` tactic is enough to finish the proof.

All other lemmas relating `ConsistentInit` and state operations are given in the following list:

```
same_map_consistent     ConsistentInitupdate
ConsistenInitPushNew    ConsistentInitPushInt
ConsistentInitPushVar   ConsistentInitPop
ConsistentInitIntTop
```

3.3 Relating `ConsistentInit` and substitution

All operations handled in the previous section have a local effect in the state. Either the top of the stack or a single variable is affected by the state modification. For substitution, it is another matter, as the modification is pervasive through the whole machine state.

As described in section 2.11, checking the consistency requires verifying that there exist a corresponding value for any uninitialized type. To verify that `ConsistentInit` is preserved through substitution, we assume that we are working with an arbitrary uninitialized type and an uninitialized object that are substituted away, given by a class type τ, an address i, an uninitialized value b; an assignment of types to variables F, a stack of types S, an assignment of values to variables f, a stack of values s, that we are substituting the type $type = (\texttt{un\_type}\ \tau\ i)$ with a new target type t in F and S and that we are substituting the value $val = (\texttt{un}\ \tau\ i\ b)$ with a new value v. We can also assume that F correctly describes the types of variables in f and S correctly describes the types in s (the last two assumptions are called `type_map_f` and `type_stack_s` in our development). Last, we assume that the target type for the substitution, t, is not of the form $(\texttt{un\_type}\ \tau''\ i'')$ (this assumption is called `not_new_un_type` in our development).

The proof divides the problem into two parts. The first part is for the case when the type that is substituted away is the uninitialized type under scrutiny. The lemma `corresponds_subst_away` expresses that any value will qualify, since the type under scrutiny disappears in the substitution process.

`corresponds_subst_away`:
$\forall b''.(\texttt{corresponds}\ \tau\ i\ b\ F\ S\ f\ s) \Rightarrow$
$(\texttt{corresponds}\ \tau\ i\ b''\ (\texttt{subst\_t}\ F\ type\ t)(\texttt{subst\_stk\_t}\ S\ type\ t)$
$(\texttt{subst\_v}\ f\ val\ v)(\texttt{subst\_stk\_v}\ val\ v))$

Proving this lemma actually relies on two extra lemmas, one for variables and one for stacks.

The second part is for the case when the type under scrutiny is different from the type that is substituted away. In that case, let us write $type' = (\texttt{un\_type}\ \tau'\ i')$ the type under scrutiny and $val' = (\texttt{un}\ \tau'\ i'\ b')$ the uninitialized value such that the following proposition holds:

$$(\texttt{corresponds}\ \tau'\ i'\ b'\ F\ S\ f\ s).$$

and let us assume that $type \neq type'$ (this assumption is called `not_type_eq` in our development). It is not possible that $val' = val$, otherwise one would have $type' = type$. thus, the value val' won't be substituted away. Thus, the value before the substitution still qualifies as after the substitution. this is summarized in the following lemma.

`corresponds_subst_diff`:
(`corresponds` $\tau'\ i'\ b'\ F\ S\ f\ s$) $\Rightarrow$
(*corresponds* $\tau'\ i'\ b'$ (`subst_t` $F\ type\ t$)(`subst_stk_t` $S\ type\ t$)
(`subst_v` $f\ val\ v$)(`subst_stk_v` $val\ v$))

The main lemma expressing that `ConsistentInit` is preserved through substitution can then be established, with the following statement:

`ConsistentInitSubst`:
(`corresponds` $\tau\ i\ b\ F\ S\ f\ s$) $\Rightarrow$ (`ConsistentInit` $F\ S\ f\ s$) $\Rightarrow$
(`ConsistentInit` (`subst_t` $F\ type\ t$)(`subst_stk_t` $S\ type\ t$)
(`subst_v` $f\ val\ v$)(`subst_stk_v` $s\ val\ v$))

For a good understanding of this statement, you need to remember that t is not of the form (`un_type` $\tau''\ i''$), $type$ is (`un_type` $\tau\ i$), and val is (`un` $\tau\ i\ b$).

3.4 The main lemmas

The main purpose of the one-step soundness theorem is to establish that being well-typed ensures some invariant through every step of execution.

`one_step_soundness`:
(`iwt_jvml` $P\ F\ S\ bound$) $\Rightarrow \forall pc.pc \leq bound \Rightarrow \forall pc', f, f', s, s'.$
(`istep` $P\ \langle pc, f, s\rangle\ \langle pc', f', s'\rangle$) $\wedge$ (`soundness_invariant` $F\ S\ f\ s\ pc$) $\Rightarrow$
(`soundness_invariant` $F\ S\ f'\ s'\ pc$) $\wedge\ pc' \leq bound.$

In this statement, (`soundness_invariant` $F\ S\ f\ s\ pc$) stands for:

(`soundness_invariant` $F\ S\ f\ s\ pc$) $\equiv$
(`type_stack` $s\ (S\ pc)$) $\wedge$ (`type_map` $f\ (F\ pc)$)$\wedge$
(`ConsistentInit` $(F\ pc)\ (S\ pc)\ f\ s$)

The relation (`iwt_jvml` $P\ F\ S\ bound$) is an inductive relation that actually stands for

$$\forall i.(\texttt{cg\_reachable}\ 0\ i) \Rightarrow (\texttt{iwtstep}\ P\ F\ S\ bound\ i).$$

As we already mentioned, `iwtstep` is itself an inductive relation with 10 constructors. Replacing (`iwt_jvml` $P\ F\ S\ bound$) with its equivalent and then producing the 10 cases of this statement corresponding to the 10 possible constructors (that is, to the 10 possible instructions in the language `jvmli`) is performed using the case theorems provided with `iwt_jvml` and `iwtstep`, through the `Elim` commands of the Coq system.

Each of the 10 cases has the following approximate form:

$\forall x_1, \ldots x_k.(P\ i) = inst \Rightarrow Q_1 \cdots Q_l \Rightarrow$
$(\texttt{istep}\ P\ \langle pc, f, s\rangle\ \langle pc', f', s'\rangle) \wedge (\texttt{soundness\_invariant}\ F\ S\ f\ s\ pc) \Rightarrow$
$(\texttt{soundness\_invariant}\ F\ S\ f'\ s'\ pc) \wedge pc' \leq bound.$

The propositions $Q_1, \ldots, Q_l$ correspond to the premises of the constructor of `iwtstep` for the instruction $inst$. The relation $(\texttt{istep}P\ \langle pc, f, s\rangle\ \langle pc', f', s'\rangle)$ is itself inductive with 10 constructors (there is no constructor for `halt`, but there are 2 for `if0`). Using the elimination theorem for this relation, we also get 10 cases for each of the previous 10 cases. In all this makes 100 cases: it is necessary to avoid handling all these cases manually.

3.4.1 A proof command for contradictory cases

We perform the first case-reasoning step and for each of the ten cases it produces, we introduce manually the quantified variables and premises that appear, thus making sure we know the name given to the hypothesis of the form $(P\ i) = inst$. Let us say this name is H_1.

A this point, we have in the context one hypothesis with the form:

$$(\texttt{istep}\ \langle pc, f, s\rangle\ \langle pc', f', s'\rangle) \wedge (\texttt{soundness\_invariant}\ F\ S\ f\ s\ pc)$$

Let us say this hypothesis is named H_2. To break down the conjunction, we can apply the following command (assuming the names H_2' and H_2'' are not already in use).

$$\texttt{Elim}\ H_2\texttt{; Clear}\ H_2\texttt{;Intros}\ H_2'\ H_2''$$

We can now use the proof command `Inversion` H_2' to perform a proof by cases according to the induction principle associated to the proposition `istep`. This command has to be used instead of the regular induction command `Elim`, to make sure the facts known about the arguments of `istep` in H_2' are not lost in the process [7]. In our case, the `Inversion` command generates as many cases as there are constructors in the definition of `istep`. For each of these cases, the command introduces all the hypotheses generated by the proof by cases in the context. At this point, there are many cases where the context contains two contradictory hypothesis, one named H_1 that states $(P\ i) = inst$, the other with an unknown name that states $(P\ i) = inst'$, where $inst$ and $inst'$ are built with different constructors.

To avoid handling all the cases by hand, we need to find a proof command that will work for all of them without a variation. The technique we use relies on the following auxiliary theorem:

$$\texttt{equal\_discr} : \forall A : Set; a, b : A.a = b \Rightarrow \forall c : A.a = c \Rightarrow\ c = b \Rightarrow False$$

Our first step is to replace the goal with the $False$ proposition, with the following command:

$$\texttt{Cut}\ False\texttt{;[Intro contradiction;Elim contradiction|Idtac]}$$

If the initial goal is C, then the command `Cut` $False$ generates two goals, where the first one is of the form $False \Rightarrow C$ and the second one is of the form $False$. The command

```
Intro contradiction;Elim contradiction
```

simply solves the first goal. The second goal is left unchanged by the proof command `Idtac`.

Now, we want to use the theorem `equal_discr` with the first two premises instantiated with the two incompatible hypotheses in the context. One of these hypotheses is easy to find, since it is referred to by a name we know: H_1. Instantiating `equal_discr` is easily done using the following command:

`Generalize (equal_discr ? ? ?` H_1`); Intros Hstep; Clear` H_1

At this point `Hstep` has the following statement:

$$\forall c : A.(P\ i) = c \Rightarrow\ c = inst \Rightarrow False$$

Two of the values mentioned in `equal_discr` have been instantiated. To find the other, we rely on existential variables created by `EApply` and the proof search performed by `EAuto`, which will instantiate the existential variables. The command we apply is:

```
EApply Hstep
```

The two goals that are generated by this command have the following form:

$$(P\ i) = ?n$$

$$\neg ?n = inst$$

In these goals $?n$ represents an *existential variable*, a variable whose value still remains to be precised. Further proof commands may proceed and constrain this variable to some value. In our case, if we use the command `EAuto` on the first goal, it will first look in the context to find if there is a candidate equality to prove this goal. Actually, there is now only one such hypothesis, because H_1 has been removed from the context when `Hstep` was created. This constrains that $?n$ is `inst'` and resolves the first goal.

Once the first goal has been solved, we can use the command `NormEvars` to propagate the constraints found on $?n$ to the second goal. In this manner, the second goal is transformed into the following one:

$$\neg inst' = inst$$

All the contradictory cases are those where $inst'$ and $inst$ are built with different constructors. In these cases, we can finish proving this goal with the proof command called `Discriminate`.

This concludes our systematic approach to proving the contradictory cases arising in proofs that perform an induction on the proposition `iwtstep` and a case by case analysis on the proposition `istep`. To avoid retyping the proof text in all cases, we have defined a tactic macro that expresses the better part of the reasoning presented here:

```
Tactic Definition DiscrCaseEq [$hyp] :=
[<:tactic:<
Try ((Cut False;
      [Intros DiscrCaseEq_reserved_name;
       Elim DiscrCaseEq_reserved_name |
       (Generalize
            (equal_discr ? ? ? $hyp);
            Clear $hyp;
        Intros
           DiscrCaseEq_reserved_name1;
        EApply
           DiscrCaseEq_reserved_name1);
           [EAuto | NormEvars; Idtac]]);
        Discriminate) >>].
```

In this command the names:

$$\texttt{contradiction}, \texttt{Hstep} \text{ and } H_1$$

have been respectively replaced with

`DiscrCaseEq_reserved_name`, `DiscrCaseEq_reserved_name1`, and `$hyp`.

3.4.2 Proving the invariant: the easy cases

After using our tactic macro `DiscrCaseEq`, the only goals that remain have a form close to:

$$(\texttt{soundness\_invariant}\ F\ S\ f'\ s'\ pc')$$

But f', s', and pc' have been replaced by their value according to the operational semantics of the instruction $inst$. In most cases pc' is simply $pc+1$ where pc is the program counter before executing the instruction. The cases are easy when f' and s' are obtained through simple operations from f and s, the values before executing the instruction. Most of the work has to be done around modifications of the stack, because the virtual machine is stack oriented.

The first easy case is when s' is simply obtained by adding a new value on top of the stack, for the instructions `load`, `push0`, `new`. In that case, proving that the new stack is well-typed is simply a matter of applying the relevant constructor of the inductive proposition `type_stack`:

`type_cons`:
$\forall v, t, s, S.(\texttt{type\_value}\ v\ t) \Rightarrow (\texttt{type\_stack}\ s\ S) \Rightarrow$
$(\texttt{type\_stack}\ (\texttt{cons}\ v\ s)(\texttt{cons}\ t\ S))$

The fact $(\texttt{type\_stack}\ s\ S)$ is simply derived from the assumption that the soundness invariant holds for the state before executing the instruction. The fact $(\texttt{type\_value}\ v\ t)$ is

usually derived in a simple manner from the premises that were obtained from the induction reasoning on `iwtstep` and `istep`.

The second easy case is when s' is actually obtained by removing a value from the top of the stack. In that case, there are premises to express that s, the stack before executing the instruction, is equal to (`cons` v s') and (S pc) is equal to (`cons` t (S pc')) and we can establish the fact:

$$(\texttt{type\_stack}\ (\texttt{cons}\ v\ s')(\texttt{cons}\ t\ (S\ pc'))) \tag{1}$$

The fact (`type_stack` s' (S pc')) is then simply obtained by inversion: the minimality of `type_stack` with respect to its constructors implies that the fact (1) could only have been obtained with the constructor `type_cons` and the premises of this constructor are necessarily true.

As far as the variables are concerned, most instructions don't modify their values and the theorem `same_type_map` makes it easy to handle these instructions quickly. The `store` instruction does modify a variable, but here the modification is a simple `update` operation and the proof is quickly established from the theorem `update_type`.

Of course, in all these cases, there are some difficult proofs to show that `ConsistentInit` also holds for the new state, but these proofs are taken care of by the theorems listed in section 3.2.

3.4.3 Proving the invariant for `init`

Proving the invariant for the `init` instruction is slightly more complicated because the modifications performed in memory when this instruction is executed are described by a pervasive change, based on `subst` and `subst_stk`.

A first part of the invariant to preserve is to ensure that the variable map and the stack are still well typed by (F $pc+1$) and (S $pc+1$) after the substitution this is given by the following two lemmas:

`type_map_subst_un`:
(`type_value` v t) $\Rightarrow$ (`type_map` f F) $\Rightarrow$
(`corresponds` σ j b F (`cons` (`un_type` σj) α) f (`cons` (`un` σ j a) s)) $\Rightarrow$
(`type_map` ($\texttt{subst}_\texttt{v}$ f (`un` σ j a) v)($\texttt{subst}_\texttt{t}$ F (`un_type` σ j) t))

`type_stack_subst_un`:
(`type_stack` s S) $\Rightarrow$ (`type_value` v t) $\Rightarrow$ (`StackCorresponds` t' i b S s) $\Rightarrow$
(`type_stack` ($\texttt{subst\_stk}_v$ s (`un` t' i b) v)($\texttt{subst}_\texttt{s}\texttt{tk}_t$ S (`un_type` t' i) t))

Proving that *ConsistentInit* is preserved through substitution operations is done using the lemma `ConsistentInitSubst` that we have already presented.

A second source of complication in the case of the `init` instruction comes from the use of dependent types, which restrict the way we can perform rewrite operations in the formulas we consider.

The proof start with the assumption that the program at line `pc` has been successfully type-checked using the rule `wt_init`. Because of this there exist a collection of values:

$$\sigma : T, j : ADDR, \alpha : (\texttt{list jtype})$$

such that the following properties hold:

$$(P\ pc) = (\texttt{init}\ \sigma)$$

$$(S\ pc) = (\texttt{cons}\ (\texttt{un\_type}\ \sigma\ j)\ \alpha)$$

$$(S\ pc + 1) = (subst\ \alpha\ (\texttt{un\_type}\ \sigma\ j)\ (\texttt{obj\_type}\ \sigma))$$

$$(F\ pc + 1) \simeq\ (subst\ (F\ pc)\ (\texttt{un\_type}\ \sigma\ j)\ (\texttt{obj\_type}\ \sigma))$$

We also know that the invariant is respected for some machine state

$$(\texttt{pc}, \texttt{f}, \texttt{s})$$

with respect to `F` and `S`. From this we can deduce the existence of values

$$a_0 : (\texttt{uninitialized\_value}\ \sigma\ j), \texttt{s}' : (\texttt{list value})$$

such that the following property holds (among others):

$$\texttt{s} = (\texttt{cons}\ (\texttt{un}\ \sigma\ ja)\ s')$$

We also know that one step of execution is applied on this state, and using our tactic `DiscrCaseEq` as in the simpler cases we can show that only the constructor `js_init` of the `istep` could have been used. From this we deduce that there exist values

$$t : T, j' : \texttt{ADDR}, a' : (\texttt{uninitialized\_value}\ t\ j'), s'' : (\texttt{list value})$$

such that the following properties hold (among others):

$$(P\ pc) = (\texttt{init}\ t)$$

$$s = (\texttt{cons}\ (\texttt{un}\ t\ j'\ a')\ s'')$$

From these equalities, we should be able to deduce that t and σ are equal, and using rewriting make sure only one name is used for the value they represent. However, rewriting does not work easily here, because if $(\texttt{un}\ t\ j'\ a')$ is well-typed, then $(\texttt{un}\ \sigma\ j'\ a')$ is not well-typed. The solution to this problem is to perform rewriting while the values that have a dependent type are still universally quantified.

For instance, if we have a property $P : value \rightarrow Prop$ and we want to show the following goal:
$\forall \sigma, \sigma' : T. \sigma = \sigma' \Rightarrow$
$\quad (\forall b : (initialized\_value\ \sigma).(P\ (obj\ \sigma\ b))) \Rightarrow$

$\forall b' : (initialized\_value\ \sigma').(P\ (obj\ \sigma\ b'))$
We cannot simply introduce all variables and premises and then rewrite σ' into σ in the goal $(P\ (obj\ \sigma'\ b'))$, because the goal $(P\ (obj\ \sigma\ b'))$ is not well typed: the type of b' has not been rewritten. Rewriting is possible but the type of b' and the expression $(obj\ \sigma'\ b')$ have to be rewritten at the same time. This happens if one introduces only σ, σ', the equality, and the other hypothesis before rewriting. At that moment the goal has the form:

$$\forall b' : (initialized\_value\ \sigma').(P\ (obj\ \sigma'\ b'))$$

Because we have to be very careful about the respective order of variable introductions and rewrites, the proof text is more intricate to write, and less work is performed by general tactics, which usually ignore this kind of fine details.

3.5 The other main theorems

The soundness result described in [11] goes beyond one-step soundness. The objective is to prove that execution will proceed normally until it reaches a halt statement if the whole program is well-typed. The one-step soundness theorem is instrumental in showing that if execution happens for a well-typed line and the state before execution is consistent with the typing information, then the state after execution will also be consistent with the typing information.

The second important theorem is the `progress` theorem. It expresses that if we are in a state that is consistent with the typing information, then execution does happen, that is, there exists a new state that will be reached through execution. This theorem has the following statement:

`progress`:
$\forall P, bound, F, S, f, s, pc,$
$(\texttt{cg\_reachable}\ 0\ pc) \Rightarrow$
$(\texttt{iwt\_jvml}\ P\ F\ S\ bound) \Rightarrow (\texttt{soundness\_invariant}\ F\ S\ f\ s\ pc) \Rightarrow$
$\neg(P\ pc) = \texttt{halt} \Rightarrow$
$\exists pc', f', s'.(smaller\ pc'\ bound) \wedge (\texttt{istep}\ P\ \langle pc, f, s\rangle\ \langle pc', f', s'\rangle)$

This proof is quite simple. We show that all the premises needed for execution are ensured by well-typing and the soundness invariant. Still we had to assume the existence of two functions `new_uninitialized` and `new_initialized` , which can always return a new values for object creation (instruction `new`) or initialization (instruction `init`).

To have correct *multi-step* execution, it is also necessary to express what multi-step execution is. We do this by introducing a relation `multi-step`, described as an inductive property to be the transitive closure of `istep`. Based on this, we have the following statement for final `soundness` theorem:

`soundness`:
$\forall P, bound, F, S.$
$(\texttt{iwt\_jvml}\ P\ F\ S\ bound) \Rightarrow$

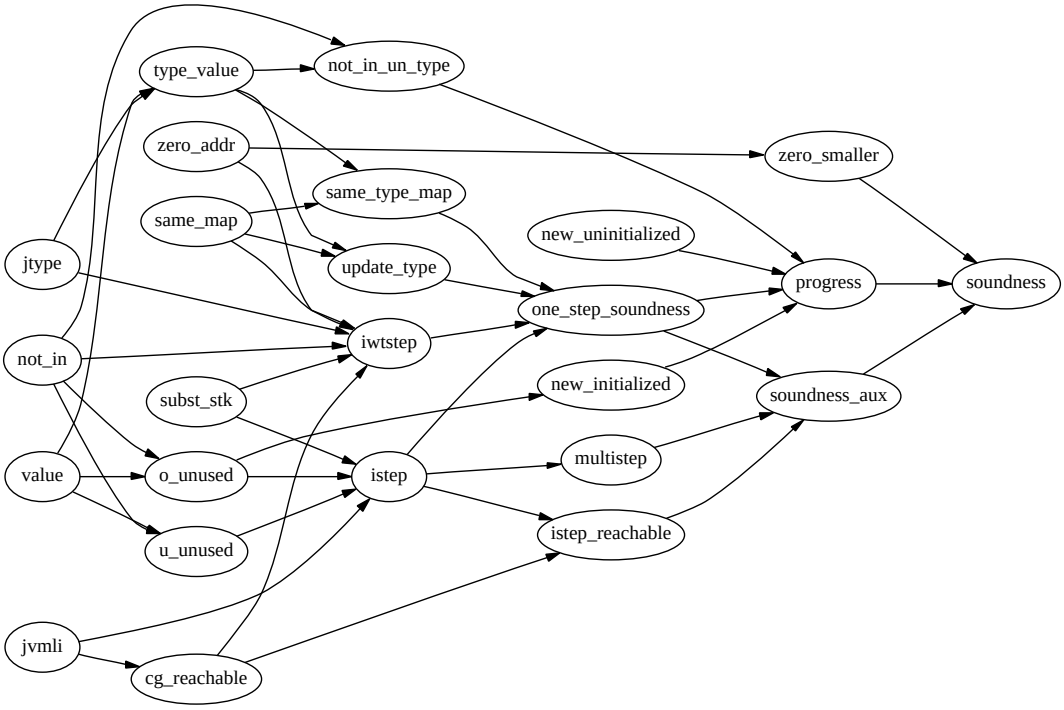

Figure 2: Proof structure for the progress and soundness theorems

$\forall pc, f_0, f, s, s'.$
$(\texttt{multistep}\ P\ \langle 0, f_0, \texttt{nil}\rangle\ \langle pc, f, s\rangle) \Rightarrow$
$\neg(\exists st.(\texttt{istep}\ P\ \langle pc, f, s\rangle\ st)) \Rightarrow$
$(\texttt{soundness\_invariant}\ F\ S\ f_0\ \texttt{nil}\ 0) \Rightarrow$
$\quad (P\ pc) = \texttt{halt} \wedge (\texttt{type\_stack}\ s\ (S\ pc))$

The structure of the proofs is sketched in figure 2. One important difference with the theorems from [11] is that we have a weaker notion for a program to be well-typed. In our sense, the program is well-typed if all addresses reachable from address zero are well-typed, but we do not impose that dead code be well-typed. Still, this weaker notion of well-typed programs is strong enough to get a similar soundness result.

4 Describing an effective algorithm

The well-typedness relation has the form $(\texttt{iwt}\ F\ S\ P)$ and can be interpreted as *P is well typed with respect to the type information given by F and S*. In this sense, the specification of `iwt` does not describe an algorithm that takes P and returns a yes/no answer. The question that remains open, and that we studied as a complement to [11], is whether one can exhibit an algorithm that takes P as an argument and returns, when it is possible, values for F and S such that $(\texttt{iwt}\ F\ S\ P)$ holds.

there are two sides to this question, and we admittedly solved only one part. We provide a function that either returns a pair (F, S) satisfying the constraints or a negative answer, expressing that it did not find such values. In case of a negative answer, we have no assurance that no such pair exists, but in case of a positive answer, we have a proof that $(\texttt{iwt}\ F\ S\ P)$ holds. Thus, this function is a sound byte-code verifier, and we have a mechanized proof, but we do not have a mechanized proof that it is complete.

Of course, a trivial byte-code verifier, that refuses every program on earth, would also satisfy these properties. Still, we hope that readers will be convinced that our byte-code verifier is not trivial.

4.1 Graph traversal algorithms

In our initial modelization, taken from [11], we have described programs as functions from a type of addresses to the type of instructions. Obviously the intent is to view addresses as natural numbers and programs as partial functions over natural numbers and the type systems given in [11] intends that all addresses where the program is defined should be well-typed. In the Coq setting, only total functions are of comfortable use, and we chose to represent programs as total functions. But a total function from natural numbers to instructions would actually represent an unrealistic infinite program. To turn around this problem, we have introduced a bound address, with the intent that the program will not be executed beyond that address.

For each program address, the information that need to be computed by the byte-code verifying algorithm has three parts: there is the map associating variables to types, the height

of the stack and the type of all locations in the stack. Concerning the stack, we are confronted with a true stack oriented machine, and there is no instruction that imposes a specific height to the stack. This height can only be deduced by propagating the information that the stack is assumed empty when starting execution at address zero through all instructions that either increment, decrement, or preserve the stack height. Thus, the relevant information will be more easily computed if the algorithm performs a complete traversal of the program control flow graph. If the program contains dead code, this code will not be checked, but after all, this code will also be harmless.

4.1.1 Agendas for graph traversal

To construct an algorithm that traverses the control flow graph while being provably terminating, we have relied on the possibility to design general recursive algorithms by basing these on well-founded definitions. we have introduced a data structure we call an `agenda`, that is meant to record the current status of all lines of the program, that is whether they have already been visited, or whether they could be visited in the next step of execution.

The basic operations available on `agenda` structures are as follows:

1. `mk_agenda: ADDR → agenda`. This creates an empty agenda, that is an agenda where no address is marked yet. The address passed as argument indicates that only addresses smaller than that one will be considered.

2. `mark_line: ADDR → agenda → agenda`. This creates a new agenda where an address has been marked as already visited.

3. `push_address: ADDR → agenda → agenda`. This creates a new agenda where an address is recorded as submitted for a future visit.

4. `next_address: agenda → (partial ADDR)`.

We want to restrict our study only to agendas that have been constructed using legitimate calls to `mk_agenda`, `mark_line`, and `push_address`. For this we define a property `acceptable` on agendas that describes just these agendas, using the following inductive definition:

```
Inductive acceptable: agenda ->Prop :=
 acc1: (i:ADDR)(acceptable (mk_agenda i))
| acc2: (i:ADDR;a:agenda)
          (acceptable a)->(acceptable (mark_line i a))
| acc3: (i:ADDR;a:agenda)
          (acceptable a)->(acceptable (push_address i a))
.
```

The returned type for the function `next_address` indicates that this function is only partial. This function should have the property that it only returns a submitted value but

not one that is marked (theorem `next_address_not_marked`). For this reason, there may be cases where no value can be returned.

Intuitively, termination is ensured by the fact that there are only a finite number of addresses in the program. Here we express this by exhibiting a relation, `more_marked`, that is well-founded. An extra constraint on `next_address` is that if it returns a value i, then (`mark_line` i a) and a are related by `more_marked` (theorem `more_marked_next`).

All the properties of the address returned by the function `next_address` can be summarized using a more qualified type, defined using the following inductive definition:

```
Inductive optional_reachable_address[a:agenda]: Set :=
 ora_success:
  (i:ADDR)
   (more_marked (mark_line i a) a) ->
   (submitted a i) -> ~ (marked a i) ->
   (optional_reachable_address a)
    | ora_fail:
     ((i:ADDR) (submitted a i) ->(marked a i)) ->
     (optional_reachable_address a) .
```

In our development, `next_address` is used to describe the informative part of a function `next_address'`, whose type is described using this new type `optional_reachable_address`:

`next_address'`: Πa:`agenda`. (`optional_reachable_address` a)

Using `next_address'` instead of `next_address` in our development will be instrumental to make the proof of termination of our graph traversal functions easier.

4.1.2 Submitted and marked lines

The type `optional_reachable_address` relies on the existence of two relations `submitted` and `marked` that describe the addresses that have been submitted using `push_address` and the addresses that have been marked using `mark_line`. These relations are practically characterized by the following properties:

`sub_mark_past`:
$\forall a : agenda.\forall i : ADDR.$(`submitted` a i) $\Rightarrow$
$\quad \forall j : ADDR.$(`submitted` (`mark_line` j a) i)
`sub_push_past`:
$\forall a : agenda.\forall i : ADDR.$(`submitted` a i) $\Rightarrow$
$\quad \forall j : ADDR.$(`submitted` (`push_address` j a) i)
`sub_push_add'`:
$\forall a : agenda.\forall i : ADDR.$(`smaller` i (`size` a)) $\Rightarrow$
$\quad$ ($submitted$ (`push_address` i a) i)
`sub_mk`:
$\forall i : ADDR.$(`submitted` (mk_agenda i) `zero_addr`)

In turn, these properties rely on the size of an agenda. Intuitively, the size of an agenda is the `ADDR` parameter that was provided at creation time to the function `mk_agenda`. The functions `mark_line` and `push_address` preserve the size property. This is expressed by a collection of properties that we do not list here.

We have similar properties for `mark_line` and `marked`. In a modular proof development, it is possible to assume all these properties and let the implementation unknown. This is the method we used, but we eventually developed a precise implementation of the `agenda` data-structure and the associated functions and proved that the properties were actually satisfied.

4.1.3 Using well-founded recursion

As we already mentioned, the termination proof for our algorithm relies on the relation `more_marked` to be well-founded, in the sense that it contains no infinite descending chain. This is expressed using a property called `more_marked_wf`:

`more_marked_wf`: (`well_founded more_marked`)

The usual practice to define a well-founded recursive function in Coq is to rely on a functional called `well_founded_induction`. This functional takes as arguments a set (here `agenda`), a binary relation on this set (here `more_marked`), a proof that this relation is well-founded (here `more_marked_wf`), a mapping from the set (here `agenda`) to sets (let's call it Φ, and a functional F that must itself receive as arguments an element x in the set and a function f of type:

$$\Pi y : \texttt{agenda}.(\texttt{more\_marked}\ y\ x) \rightarrow (\Phi\ y)$$

and should return an value of type $(\Phi\ x)$. Intuitively, the functional F represents the body of the recursive function being defined and f represents the recursive calls to the function that may occur in the body. The complicated type given to the function f expresses that recursive calls can only be made on agendas that have more marks. Now the fact that `more_marked` is well-founded ensures that infinite recursion will not occur.

In our work, the structure of the functional F will always have the same form, summarized in the following pseudo code:

1. require a next address (using `next_address`) from the current agenda, if no such next address exists, then computation is over,
2. process the line at the given address, potentially working on some auxiliary data,
3. construct a new agenda, where the address given at step 1 is marked and the address that could follow in execution are submitted.
4. call recursively the function on the new agenda and new auxiliary data, when it exists.

As can be seen here, the algorithm is basically tail recursive, so that modifications on the agenda could actually be done in place. This allows for an imperative implementation of this part of our algorithm.

Step 2 of these algorithm will fail when an error is detected. In this case, the recursion is terminated with an error result.

4.2 A two pass breakdown

The constraints to verify for each line can be broken down in two parts. The first part is *cumulative*. This kind of constraints is valid even if the information gathered at some line is incomplete, in the sense that some memory locations may still have an unknown type, but any choice of type for these locations will still respect the constraints. The second part is *restrictive*: to verify these constraints, it is necessary to know the type associated to every location in memory for the line being studied. Restrictive information constraints appear only for lines containing a `new` or `init` instruction.

We broke down the verification algorithm in two passes. The first pass accumulates information and verifies the cumulative constraints. The second pass checks that the information gathered during the first pass also verifies the restrictive constraints. No accumulation happens anymore in the second pass.

4.2.1 The first pass

Iteration usually works by progressively modifying a state. When encoding iteration in a purely functional programming style, there is no side-effect on a state and this is actually encoded by having an extra argument to the recursive function that represents the state that is modified during execution. We thus define a function that takes an agenda and a state as arguments and returns an optional state as value. When a state is returned, this means that the type verification succeeded. When no value is returned this must be interpreted as a failure: the program should be rejected. In our work, `typestruct` is the type of states and `(partial typestruct)` is the type expressing that negative answers may be returned.

We also have two functions `get_frame` and `get_stack` that return respectively the F part and S part of the information needed in the well-typedness proposition. Thus, these functions have the following types:

$$\begin{array}{lcl} \texttt{get\_frame} & : & \texttt{typestruct} \rightarrow \mathit{ADDR} \rightarrow V \rightarrow \texttt{jtype} \\ \texttt{get\_stack} & : & \texttt{typestruct} \rightarrow \mathit{ADDR} \rightarrow (\mathit{list}\ \texttt{jtype}) \end{array}$$

Using the common presentation of our traversal algorithms given in section 4.1.3, the first pass uses a type for returned value of the following form:

$$\Phi \equiv \lambda x : \mathit{agenda}.(\texttt{acceptable}\ x) \rightarrow \texttt{typestruct} \rightarrow (\texttt{partial typestruct})$$

Thus, the function f representing recursive calls has the following type:
$\forall y : \texttt{agenda}.(\texttt{more\_marked}\ y\ x) \rightarrow (\texttt{acceptable}\ y) \rightarrow \texttt{typestruct} \rightarrow$
(`partial typestruct`)

For each line we want to refine the current state to take into account the constraints imposed by the instruction at that line. Constraints are represented by a type `type_constraint`

and each instruction in the language may impose several of these constraints. Thus we have an auxiliary function called `jvmli_to_constraint_list` to map instructions to lists of constraints.

To get this clearer, let's have a look at the constraints associated to the instruction `inc`. If `inc` is the instruction at line i, then the type of variables at line i and $i+1$ should coincide, the type of stack at line i and $i+1$ should coincide, and there should be a value of type `int` on top of the stack. We represent this with three constraints of the following form:

$$(\texttt{tc\_all\_vars}\ i\ i+1)$$

$$(\texttt{tc\_stack}\ i\ i+1)$$

$$(\texttt{tc\_top}\ i\ \texttt{int})$$

Once we have constraints associated to an instruction and a line number, we need to modify the state accordingly. This is done using a function we named `add_constraint_list'` with the following type:

`add_constraint_list'`:
`typestruct` $\rightarrow$ (`list type_constraint`) $\rightarrow$ (`partial typestruct`)

The returned value is only optional, and programs are rejected at a given line when a constraint fails to be added for the instruction at line.

Once the constraints at a given line have been checked, it is important to proceed with the lines where the control can be transfered. This is done by marking the current line in the agenda (using the function `mark_line`) and submitting the lines where control can be transfered (using the function `push_address`. This marking and submitting operation is described by a function (`new_agenda`).

The complete encoding of the body functional (as described in section 4.1.3) is given in figure 3. In this function, `P` is supposed to represent the current program (as a function from addresses to instructions). Also, although this does not appear in the writing, the `new_agenda` function is actually parameterized by `P`, because the addresses to submit depend on the value `(P i)`.

The actual first pass function is the function `verifier` defined as follows:

```
Definition verifier:
  (a:agenda) (acceptable a)->typestruct->(partial typestruct) :=
   (well_founded_induction
    agenda more_marked more_marked_wf
    [t:agenda](acceptable t)->typestruct->(partial typestruct)
     verif_F).
```

To work on this function, it has proved handy to use the following *unfolding* theorem, that we have proved using the technique described in [1].

`unfold_verifier`:
$\forall t$: `typestruct`, a : `agenda`, h : (`acceptable` a).

```
Definition verif_F:
  (a:agenda)
  ((b:agenda)(more_marked b a) -> (acceptable b) -> typestruct ->
   (partial typestruct)) ->
  (acceptable a) -> typestruct ->(partial typestruct).
Refine [a, f, Hacc, t:?]
          Cases (next_address' a Hacc) of
              (ora_fail h) =>
               (Some t)
             | (ora_success i h h1 h2) =>
                Cases (add_constraint_list'
                          (jvmli_to_constraint_list i (P i)) t) of
                    None =>
                     (None typestruct)
                   | (Some t') =>
                      (f (new_agenda a i) ? ? t')
                end
          end; Auto.
Defined.
```

Figure 3: Formal encoding of the first pass

$$(\texttt{verifier}\ a\ h\ t) =$$
$$(\texttt{verif\_F}\ a\ \lambda b : \texttt{agenda}.\lambda h' : (\texttt{more\_marked}\ b\ a).(\texttt{verifier}\ b)\ h\ t)$$

The graph displaying the dependencies between the various concepts involved in the definition of the function `verifier` is displayed in figure 4. The part of this graph containing the nodes depending on `fun` actually gives a first preview of the implementation of `typestruct` and `add_constraint_list'`, based on first-order unification. The subgraph depending on `agenda`, `acceptable` is a summary of the proofs involved in ensuring the termination of the algorithm.

4.3 Defined stacks

In section 4.1, we showed that one of the important pieces of information that needs to be computed for each line is the height of the stack. When our byte-code verifier starts analyzing the program, this information is unknown for all lines but the first one. As verification progress, we will show that the height of the stack is defined for all lines that are submitted for analysis, and this is preserved as an invariant. We use the notation $(\texttt{stack\_defined}\ t\ i)$ to express that the height information is already known for line i in the state t.

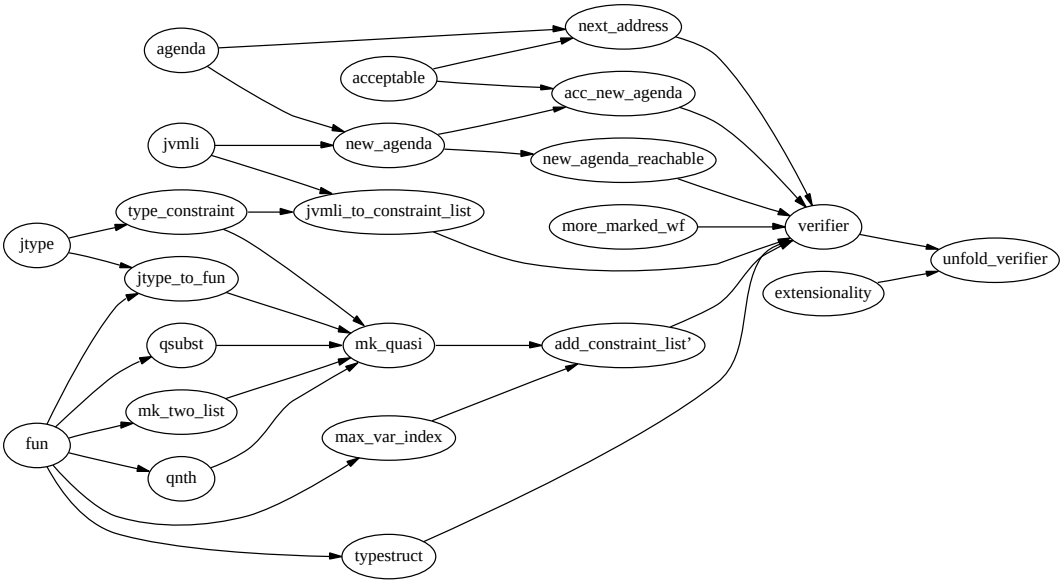

Figure 4: The verifier function main structure. *This graph exhibits a separation of concerns between agenda-related concepts, that are concerned with organizing graph traversal and ensuring its termination, and constraint application.*

We will see below that some constraints require that the stack be defined for the line being verified to be meaningful. We will also see that some constraints propagate the property of having a well-defined stack to other lines.

4.4 Cumulative Constraints

The algorithm works by successively adding constraints to the state. Once a constraint has been added, the information should be recorded, in such a way that adding the same constraint later would not fail, and not change the state anymore. We introduced a proposition `constraint_applied` that takes as arguments a constraint and a state, and expresses that this constraint has been applied on this state.

We also introduced an order between states of type `typestruct`, called `tsleq` and denoted $\preceq$ throughout this paper. This order should be verified for two states when the first one has been obtained from the second one by adding a collection of constraints:

$$\exists l.(\texttt{add\_constraint\_list}'\ l\ t) = (\texttt{Some}\ t') \Rightarrow t' \preceq t$$

Constraints are cumulative when they are stable with respect to the order $\preceq$. In this section, we are going to enumerate the five cumulative kinds of constraints that we isolated in our study of the language.

4.4.1 Untouched variables

Only two instructions modify the variables: for all other instructions, we need to express that the types of variables are the same for the current line and the next line of execution. This is done using a constraint named (`tc_all_vars` i j), where i and j are the two lines that must correspond.

The semantics of this constraint is summarized by the following property:

`tc_all_vars_semantics`:

$$\forall i, j, t, t'.(\texttt{constraint\_applied}\ (\texttt{tc\_all\_vars}\ i\ j)\ t) \Rightarrow t' \preceq t \Rightarrow F_{t'}(i) = F_{t'}(j)$$

4.4.2 Untouched stack

Similarly there is an instruction that does not modify the stack, the instruction `inc`, so that the next line must have the same type information for the stack. This constraint is expressed using a constraint named (`tc_stack` i j), where i and j are the two lines that must correspond.

The semantics of this constraint is summarized by the following property:

`tc_stack_semantics`:

$$\forall i, j, t, t'.(\texttt{constraint\_applied}\ (\texttt{tc\_stack}\ i\ j)\ t) \Rightarrow t' \preceq t \Rightarrow S_{t'}(i) = S_{t'}(j)$$

4.4.3 Fixing the top type on the stack

For several instructions, it is necessary that a given type be present on top of the stack before executing the command: the type `int` for instructions like `inc` or `if0`, the type $(\texttt{obj\_type}\ \sigma)$ for an instruction $(\texttt{use}\ \sigma)$. This is expressed using a constraint named $(\texttt{tc\_top}\ i\ \tau)$, where i is the line that must have the type τ on top of the stack.

The semantics of this constraint is summarized by the following property:

`tc_top_semantics`:
$\forall i, \tau, t, t'.(\texttt{constraint\_applied}\ (\texttt{tc\_top}\ i\ \tau)\ t) \Rightarrow$
$t' \preceq t \Rightarrow \exists \alpha.\quad S_{t'}(i) = (\texttt{cons}\ \tau\ \alpha)$

4.4.4 Popping a type off the stack

Most instruction work on the value on top of the stack. Those that use this value normally remove the argument from the stack. For types, this is expressed using a constraint named $(\texttt{tc\_pop}\ i\ j)$ where i is the address where the stack has an extra value on top of the stack present at line j.

The semantics of this constraint is summarized by the following property:

`tc_pop_semantics`:
$\forall i, j, t, t'.(\texttt{constraint\_applied}\ (\texttt{tc\_pop}\ i\ j)\ t) \Rightarrow$
$(\texttt{stack\_defined}\ t\ i) \Rightarrow$
$t' \preceq t \Rightarrow \exists \tau, \alpha. S_{t'}(i) = (\texttt{cons}\ \tau\ \alpha) \wedge S_{t'}(j) = \alpha$

4.4.5 Pushing a variable type on the stack

The instruction `load` pushes value on top of the stack, the type of this value is taken from another variable in memory. This is expressed using a constraint named $(\texttt{tc\_pop}\ i\ x\ j)$ where i is the address where the initial stack and the initial value are take, x is the variable index for the type is pushed on the stack and j is the address where the new stack appears.

The semantics of this constraint is summarized by the following property:

`tc_push_semantics`:
$\forall i, j, x, t, t'.$
$i \neq j \Rightarrow (\texttt{constraint\_applied}\ (\texttt{tc\_push}\ i\ j\ x)\ t) \Rightarrow$
$(\texttt{stack\_defined}\ t\ i) \Rightarrow$
$t' \preceq t \Rightarrow \exists \tau, \alpha. S_{t'}(j) = (\texttt{cons}\ F_{t'}(i)\ \alpha) \wedge S_{t'}(i) = \alpha$

4.4.6 Pushing a precise type on the stack

Instructions `new` and `push0` push a new value on the stack, but the type of the new value can be decided from the instruction being executed, not from a variable in memory. This is expressed using a constraint $(\texttt{tc\_push\_type}\ i\ j\ \tau)$, where τ is the type being pushed, i is the address where the instruction lies and j is the address where the augmented stack occurs.

The semantics of this constraint is summarized by the following property:

`tc_push_type_semantics`:
$\forall i, j, \tau, t, t'.$
$i \neq j \Rightarrow (\texttt{constraint\_applied}\ (\texttt{tc\_push\_type}\ i\ j\ \tau)\ t) \Rightarrow$
$(\texttt{stack\_defined}\ t\ i) \Rightarrow$
$\quad t' \preceq t \Rightarrow \exists \alpha.\quad S_{t'}(i) = \alpha \wedge S_{t'}(j) = (\texttt{cons}\ \tau\ \alpha)$

4.4.7 Setting the type of a variable

When the instruction `store` is called, the value of some variable changes, and possibly its type too (`jvml` variables are distinguished from Java variables and a `jvml` variable may be used to store several Java variables, as pleases the optimizations performed by the Java compiler). The new type for the variable is the type that was on top of the stack before the instruction was called. Only the variable affected by the store operation is modified. This is expressed using a constraint named $(\texttt{tc\_store}\ i\ j)$ where i and j are lines of execution before and after executing the store command.

The semantics of this constraint is more complex than for all the other constraints, as it affects both the variables and the stack.

`tc_store_semantics`:
$\forall i, j, x, t,$
$i \neq j \Rightarrow (\texttt{constraint\_applied}\ (\texttt{tc\_store}\ i\ x\ j)\ t) \Rightarrow$
$(\texttt{stack\_defined}\ t\ i) \Rightarrow$
$\quad \forall t'.t' \preceq t \Rightarrow \exists \alpha.\quad S_{t'}(i) = (\texttt{cons}\ F_{t'}(j)\ \alpha) \wedge$
$\qquad F_{t'}(j) = (\texttt{update}\ F_{t'}(i)\ x\ F_{t'}(j, x)) \wedge S_{t'}(j) = \alpha$

Arguably, the semantics of `tc_store` could have been made simpler. For instance, one could have only taken care of the behavior with respect with variable types, letting `tc_pop` take care of the effects on the stack.

4.5 Restrictive constraints

The constraints required for type-checking occurrences of the instructions `new` and `init` cannot be completely expressed using only cumulative constraints. For the `new` instructions, we have to express that a new type $(\texttt{un\_type}\ \sigma\ i)$ needs to be added on top of the stack, but one should also verify that this type does not occur anywhere else in memory. This negative constraint is not cumulative. If some variable has its type unknown at the time one verifies line i one cannot know whether this variable's type will be made equal to an unwanted type later in the process. For the first pass, we do not check the restrictive part of the constraint.

For the instruction `init` the situation is more complicated: it is not even possible to proceed without trying to apply a restrictive constraint, because initialization will set correct type for variable locations that are used later. Our approach here is to let the first pass

apply the restrictive constraint, leaving the second pass perform a second check to ensure that the constraint is still verified after all cumulative constraints have been applied.

The constraint for the init instruction is written ($\texttt{tc\_init}\ i\ j\ \sigma$). It can be applied only when the stack at line i has a type ($\texttt{un\_type}\ \sigma\ k$) on top. In that case the stack at line j should be the result of popping the value on top and replacing all other instances of ($\texttt{un\_type}\ \sigma\ k$) with ($\texttt{obj\_type}\ \sigma$). The same replacement should be applied on variables.

The semantics of this constraint is expressed by the following properties:

`tc_init_stack_exists`:
$\forall i, j, t, \sigma,$
$(\texttt{add\_constraint}'\ (\texttt{tc\_init}\ i\ j\ \sigma)\ t) = (\texttt{Some}\ t) \Rightarrow$
$(\texttt{stack\_defined}\ t\ i) \Rightarrow$
$\quad \exists k, \alpha.\quad S_t(i) = (\texttt{un\_type}\ \sigma\ k) \cdot \alpha$

`tc_init_frame`:
$\forall i, j, k, t, \sigma, \alpha.$
$\quad (\texttt{add\_constraint}'\ (\texttt{tc\_init}\ i\ j\ \sigma)\ t) = (\texttt{Some}\ t) \Rightarrow$
$\quad S_t(i) = (\texttt{un\_type}\ \sigma\ k) \cdot \alpha \Rightarrow$
$\quad\quad F_t(j) = (subst\ F_t(i)\ (\texttt{un\_type}\ \sigma\ k)\ \sigma)$

`tc_init_stack`:
$\forall i, j, k, t, \sigma, \alpha.$
$\quad (\texttt{add\_constraint}'\ (\texttt{tc\_init}\ i\ j\ \sigma)\ t) = (\texttt{Some}\ t) \Rightarrow$
$\quad S_t(i) = (\texttt{un\_type}\ \sigma k) \cdot \alpha \Rightarrow$
$\quad\quad S_t(j) = (subst\_stk\ \alpha\ (\texttt{un\_type}\ \sigma\ k)\ \sigma)$

Note that these statements only state properties for the structure t when it is unchanged by the operation of adding the constraint, not for possible refinements of this structure through $\preceq$.

4.6 Mapping instructions to lists of constraints

To map instructions to lists of constraints, we write a simple function that performs a case analysis on the instructions. This function is given in Coq syntax in figure 5.

5 Relying on an unification algorithm

To implement type structure and constraints, we mostly used an unification algorithm. The main advantage of this approach was to re-use an algorithm that had been provided in the contributions of the Coq system. This algorithm was developed by J. Rouyer from INRIA Nancy, and works on a simplified notion of terms called quasi terms. We use our own modified version of the algorithm, where the constants used to represent term operators have been changed from natural numbers in the initial implementation to an arbitrary type given as parameter, together with an equality test function.

```
Definition jvmli_to_constraint_list:=
[i:ADDR] [inst:jvmli] Cases inst of
 inc => (cons
          (tc_all_vars i (add1 i))
          (cons (tc_stack i (add1 i))
             (cons (tc_top i int) (nil ?))))
| pop => (cons
          (tc_all_vars i (add1 i))
          (cons (tc_pop i (add1 i)) (nil ?)))
| push0 => (cons
              (tc_all_vars i (add1 i))
              (cons (tc_push_type i (add1 i) int)
                    (nil ?)))
| (load x) => (cons
                (tc_all_vars i (add1 i))
                (cons (tc_push i x (add1 i)) (nil ?)))
| (store x) => (cons (tc_store i x (add1 i)) (nil ?))
| (if0 L) =>
     (cons
        (tc_all_vars i (add1 i))
        (cons
           (tc_all_vars i L)
           (cons
              (tc_pop i (add1 i))
              (cons (tc_pop i L)
                    (cons (tc_top i int) (nil ?)))))))
| (new sigma) =>
     (cons
        (tc_all_vars i (add1 i))
        (cons
           (tc_push_type i (add1 i)
                (un_type sigma i)) (nil ?)))
| (init sigma) => (cons (tc_init i (add1 i) sigma) (nil ?))
| (use sigma) =>
     (cons (tc_all_vars i (add1 i))
        (cons
           (tc_pop i (add1 i))
           (cons (tc_top i (obj_type sigma)) (nil ?))))
| halt => (nil ?)
end.
```

Figure 5: Mapping instructions to lists of constraints

The type of quasi terms contains pairs of quasi terms, function terms, where an operator is coupled with a quasi term, variables and constants indexed with operators. It is described with the following inductive definition:

```
Inductive quasiterm:Set:=
V:var->quasiterm
|C:fun->quasiterm
|Root : fun->quasiterm->quasiterm
|ConsArg  : quasiterm->quasiterm->quasiterm.
```

The type `fun` is actually given as a parameter. We are going to use this type to denote list constructors and types.

We use a pair of quasi terms to represent all the knowledge gathered about the functions F and S. Ultimately, the pair's first element should be a list of lists, where each list gives the type information for all variables at the corresponding line in the program. The pair's second element should also be a list of list, but here each lists give the type information for the stack at the corresponding line.

In the initial state, the pair's first element is simply a quasiterm variable, to express that we know nothing about the variables' initial type. The pair's second element is the term representing a list, the first element of which is an empty list, and the rest of which is a quasiterm variable, to express that we know that the initial stack is empty and that we know nothing about stacks on the other lines.

5.1 Encoding lists and types

We want to use quasi terms to represent partially known information about bi-dimensional arrays of types. To represent bi-dimensional arrays, we encode them as lists of lists. To represent types, we use constant quasi terms and we let the operator express what type is actually represented. To achieve this result, we define the type `fun` with the following inductive definition:

```
Inductive fun: Set :=
  fcons: fun
| fnil : fun
| fint : fun
| otype: T -> fun
| utype: T -> ADDR -> fun.
```

The constructors `fcons` and `fnil` are used for representing list constructions. The other three constructors are used to represent types.

Empty lists are represented with the term (`C fnil`). Lists with a first element represented by t_1 and the rest represented by t_2 are represented by the term:

$$(\texttt{Root fcons } (\texttt{ConsArg } t_1\ t_2))$$

To make this encoding more easily available we defined a function `qcons`, so that this term can also be written (`qcons` t_1 t_2).

For all constraints except the constraint `tc_init`, applying the constraints simply means unifying the state with a term that expresses which locations have to be equal. For instance, for the constraint (`tc_all_vars` i j), when $i < j$, we use the term:

$$\begin{array}{l}(\texttt{ConsArg}\ (\texttt{qcons}\ V_{k+2}\ (\texttt{qcons}\ \cdots\ (\texttt{qcons}\ V_{k+i+1}\ (\texttt{qcons}\ V_k\ (\texttt{qcons}\ V_{k+i+2}\cdots \\ \qquad (\texttt{qcons}\ V_{k+j}\ (\texttt{qcons}\ V_k\ V_{k+j+1}))\cdots) \\ \quad V_k+1)\end{array}$$

This is the encoding of a list containing variables that all pairwise distinct, except that the variable at positions i and j are equal. The tail of the list after position j is also represented by a variable (V_{k+j}). The index k is chosen so that none of the variables occurring in the term representing the constraint occur in the term on which the constraint is applied, to avoid imposing useless equalities.

Constructing this kind of list, with all members being pairwise distinct variables and only two specified lines represented by the chosen terms, is done by a recursive function we call `mk_two_list`, the term constructed for the constraint (`tc_all_vars` i j) can also be represented by the expression:

$$(\texttt{ConsArg}\ (\texttt{mk\_two\_list}\ V_{k+1}\ V_{k+1}\ (j-i)\ i\ k)\ V_k)$$

In a similar manner, we also provide a function `place_one_list` that constructs a term of the following form:

$$\begin{array}{l}(\texttt{place\_one\_list}\ t\ i\ k) = \\ \quad (\texttt{qcons}\ V_k\ (\texttt{qcons}\ \cdots\ (\texttt{qcons}\ V_{k+i-2}\ (\texttt{qcons}\ t\ V_{k-1}))\cdots))\end{array}$$

This constructs a list with all members being pairwise distinct variables except for the i^{th} element that is set to the specified value t and the tail after the i^{th} is also represented by a variable.

With these functions, the constraint (`tc_stack` i j) is represented by the following term, when $i < j$:

$$(\texttt{ConsArg}\ V_k\ (\texttt{mk\_two\_list}\ V_{k+1}\ V_{k+1}\ (j-i)\ i\ k))$$

The constraint (`tc_top` i τ) is represented by the following term:

$$(\texttt{ConsArg}\ V_k\ (\texttt{place\_one\_list}\ (\texttt{qcons}\ (\texttt{C}\ (\texttt{otype}\ \tau))\ V_{k+1})\ k+1))$$

The constraint (`tc_push` i x j) is represented by the following term, when (i<j):

$$\begin{array}{l}(\texttt{ConsArg}\ (\texttt{place\_one\_list}\ (\texttt{place\_one\_list}\ V_k\ x\ (k+2))\ i\ (k+x+3)) \\ \quad (\texttt{mk\_two\_list}\ V_{k+1}\ (\texttt{qcons}\ V_k\ V_{k+1})\ (j-i)\ i\ (k+x+i+4)))\end{array}$$

All these constraint representations are collected in the behavior of a function we have called `mk_quasi`.

5.2 Second pass

The second pass is also a graph traversal, using exactly the same `agenda` structure to control the progress of the algorithm. Before starting this pass, we make the typing state closed, by replacing all remaining logical variables with the type `top`. In this manner, constraints are only checked on the structure, but they are not added anymore. Aside from checking the constraints for the instructions `new` and `init`, this pass also checks basic properties of branching constructs: that non-halt instructions are always found at an address strictly smaller than the `bound` address, that branching instructions always branch to an instruction smaller than the `bound` address, and that branching instructions don't branch to themselves.

6 Proving the soundness of the verifier

6.1 First pass soundness

To express the soundness of the algorithm's first pass, we introduced a weaker type system, where the restrictive information has been removed. This type system is described with the same constructors as the type system proposed in [11], with only two constructors changed. The constructor for the instruction `new` becomes the following one:

```
wt_new':
 (sigma:T)
 (P i)=(new sigma) ->
 (same_map jtype (F (add1 i)) (F i)) ->
 (S (add1 i))=(cons (un_type sigma i) (S i)) ->
(smaller (add1 i) bound) ->(iwtstep' P F S bound i)
```

The main difference between this constructor and the one for `iwtstep` is that we do not check that the type that is created at that line is not already present in memory (a predicate `not_in` has been removed).

The constructor for `init` in this new type system just accepts any line containing an `init` instruction.

The proof of soundness for this pass is then simply expressed by saying that if the `verifier` succeeded on some program, then all reachable lines from the start address satisfy the weaker type system with respect to the type information returned by the verifier.

Here there is an invariant to verify as the verifier progresses on new agenda and new `typestruct` states. In this invariant, we have to express that :

1. the `agenda` structure is `acceptable` in the sense that it is the result of a succession of `push_address` or `mark_line` operations from an initial state created with `mk_agenda`,
2. the size of the agenda is the same as the `bound` address (modulo conversion through the injection from addresses to natural numbers `a_to_nat`,
3. all lines marked in the agenda already satisfy the type system,

4. all lines reachable in one step of execution from a line that is already marked are submitted: either they are candidates for a next step or they are themselves already marked,

5. all lines are that are submitted in the agenda have their stack well-defined in the current `typestruct` state.

Establishing that this invariant is maintained throughout execution relies on the following facts.

1. The only operations done to obtain the agenda for the recursive call are described by the function `new_agenda`. In all cases `new_agenda` only performs calls to `mark_line` and `push_address`. This is expressed in the theorem `new_agenda_acceptable`.

2. The size of agendas is shown to be unchanged through `mark_line` and `push_address` operations. This is expressed in a theorem that we named `new_agenda_keep_size`.

3. Each time a line is processed, only this line is added among the marked lines. We just show that the success of adding the constraints corresponding to the instruction at that line is enough to ensure that the type system is satisfied at that line. Now, since we only look at cumulative constraints in this pass, the constraints will remain satisfied for all successive `typestruct` states constructed during the pass, because they are only more precise (as described by the order $\preceq$ on `typestruct` states). This is expressed in theorem `sound_verifier_aux_line`

4. We show that `new_agenda` is correctly constructed in the sense that it really marks for execution those lines that are reachable from the current line. This is expressed in the theorem `new_agenda_correct`.

5. We show that adding the constraints corresponding to each line ensures that the stack will be well defined in the `typestruct` state for all lines that are reachable in one step.

The fact that the invariant is obtained for the final agenda and the final typing state is expressed in the theorem that we called `verifier_sound_main`. The structure of this proof is displayed in figure 6.

6.2 The constraint interface

Instead of directly using the encoding of constraints with quasi terms to prove that the success of constraint applications ensures that the type system is satisfied for each line, we organize the proof to use only the theorems that we stated about the semantics of each type constraint. For each instruction, we need to prove that the constraints provided for that instruction by the function `jvmli_to_constraint_list` are strong enough to ensure that all the premises for the corresponding constructor in the weaker type system are satisfied. We do this in ten different theorems, for the ten different instructions. The structure of this proof is shown in figure 7.

Figure 6: Structure of the proof of soundness. The proof of `verifier_sound_aux_line` is detailed in another graph.

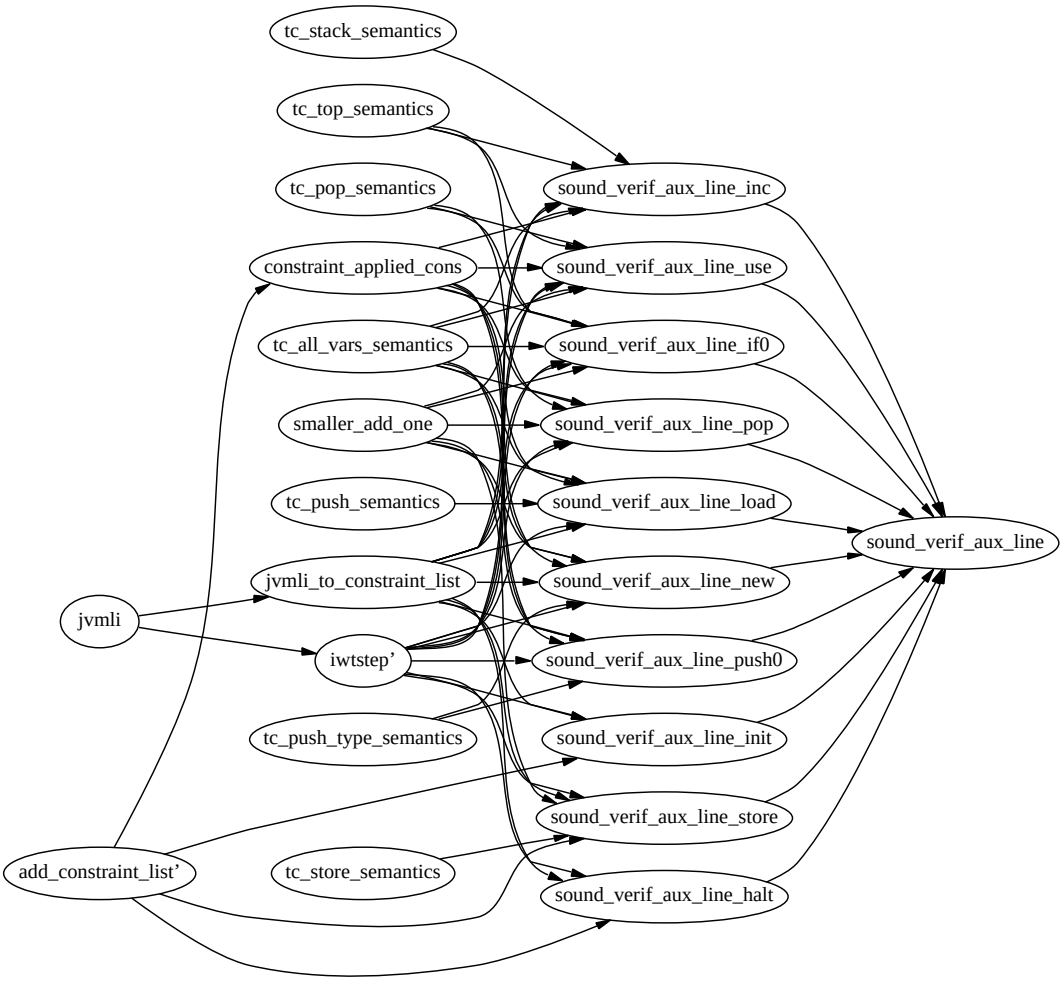

Figure 7: Structure of the soundness proof for the first pass: the theorems for each line.

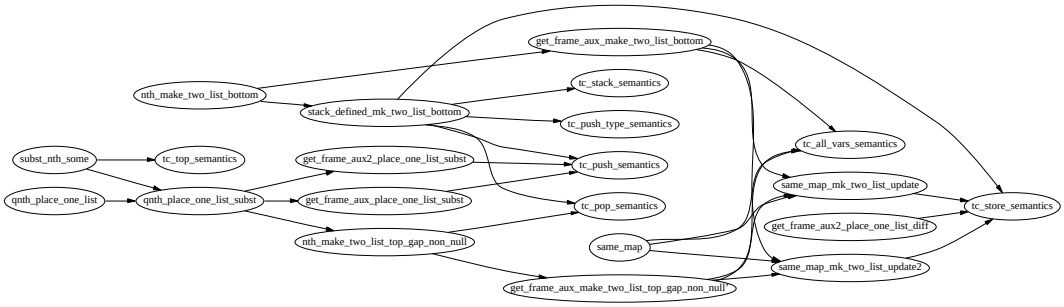

Figure 8: Proving the correctness of constraints encoding. The constraints `tc_all_vars` and `tc_store` have a special status because they act on the variables. The proof for the constraint `tc_init` is not displayed in this graph.

It is then necessary to prove that each of the «semantic» theorems is actually satisfied, by checking the consistency between the quasi terms constructed in `mk_quasi` and the functions `get_frame` and `get_stack` used to interpret the `typestruct` state as a couple F, S used in the typing judgment. The structure of this proof is described in figure 8.

For these proofs, we have to reason on objects in the type `quasiterm`, provided for the unification algorithm, which encode lists, and lists of lists. To reason on these objects, it is sensible to perform induction proofs that reflect the structure of lists (with two cases) rather than induction proofs that reflect the structure of `quasiterm` values. To make these proofs easier we have introduced and proved a derived induction principle with the following statement:

`qlist_ind`:
$\forall P : (\texttt{quasiterm fun}) \rightarrow \texttt{Prop}.$
$(\forall x : (\texttt{quasiterm fun}).(\forall t_1, t_2.\neg x = (\texttt{qcons } t_1\ t_2)) \Rightarrow (P\ x)) \Rightarrow$
$(\forall t_1, t_2 : (\texttt{quasiterm fun})(P\ t_2) \Rightarrow (P\ (\texttt{qcons } t_1\ t_2))) \Rightarrow$
$\forall t : (\texttt{quasiterm fun}).(P\ t)$

The first case does not exactly correspond to the `nil` case in an usual presentation of lists. Rather, it covers all terms that do not encode lists with at least one element. The possibility to define new induction principles, more adapted to our use of the data-structure is one of the characteristic advantages of theorem provers based on higher-order logic.

The constraint `tc_init` holds a special place. The property it ensures is not stable through type specialization as represented by the order `tsleq` ($\preceq$). This constraint also relies on an encoding of substitution on `quasiterm` data, `qsubst`. The structure of the proofs around this constraint is sketched out in figure 9.

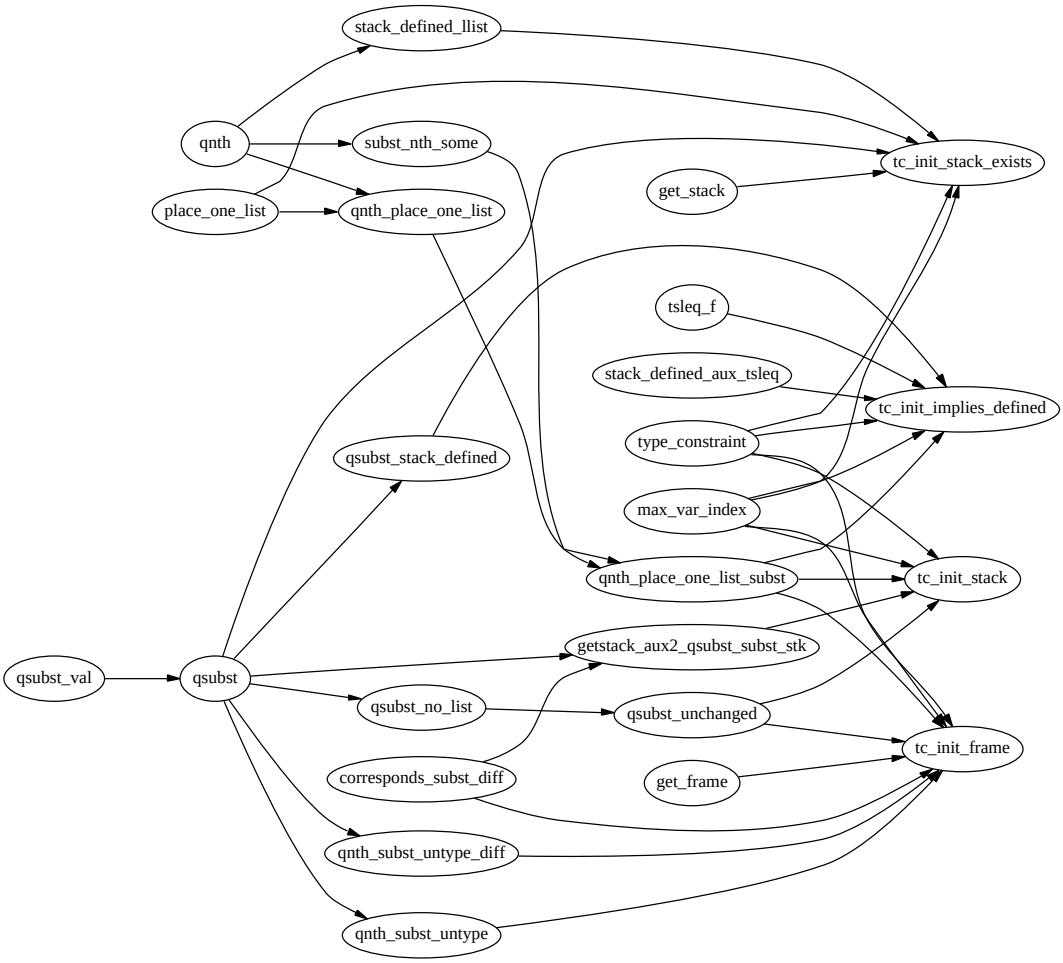

Figure 9: Structure of proofs for the constraint `tc_init`

7 Conclusion

The extraction mechanism of Coq makes it possible to obtain from this formal description a program that will run on simple examples. Still, this program is very likely to be unpractical: no attention has been paid to the inherent complexity of the verification mechanism. At every iteration we construct terms whose size is proportional to the line number being verified: in this sense the algorithm complexity is already sure to be quadratic. A strong improvement should be obtained if we decide to keep the next address closer to reach, for instance using the data-structures that G. Huet playfully named *zippers* [15]. Another direction is to avoid using unification, trying instead to use a `union-find` algorithm à la Tarjan [28], using a partition of all variables at all lines, where two variables being in the same equivalence class means they must have the same type.

Still, even if the exact representation of the typing state and constraints are likely to change to obtain a more usable verifier, we believe that the decomposition of its implementation and certification in the various phases presented in this paper is likely to remain relevant. These phases are:

1. Proving the soundness of a type system based on data not provided in the program,
2. Proving that a program building the missing data ensures that the typing constraints are satisfied,
3. Decomposing the typing constraints for each instruction into more primitive constraints that can be shared between several instructions,
4. Setting aside the constraints that may not be preserved through the refinements occurring each time a line is processed,
5. Performing a graph traversal by using an agenda structure, where lines are marked and submitted,
6. Proving an invariant on the graph traversal where the typing properties are assumed for the marked lines and where one proves that they will be obtained for all lines reachable from the submitted lines.

With a broader perspective, this development of a certified byte-code verifier, can be understood as an example of using a type-theory based proof system as a programming language in the domain of program analysis tools, with all the benefits of the expressive type system to facilitate low-error programming and re-use of other programs and data-structures, as we did with the unification algorithm of [26]. Still we have to be aware that the work presented here might still contain bugs in some sense: we have only proved that our byte-code verifier will only accept programs that behave soundly, we never proved that it will accept any programs at all. This question, and other questions pertaining to the efficiency of the verifier will be the objective of future work.

References

[1] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[2] Yves Bertot and Olivier Pons. Dependency graphs in theorem provers, 2000. To appear as INRIA Research Report, also visible as http://www-sop.inria.fr/lemme/graphs/.

[3] Ludovic Casset and Jean-Louis Lanet. How to formally specify the java byte code semantics using the b method. In *proceedings of the Workshop on Formal Techniques for Java Programs at ECOOP 99*, June 1999.

[4] Thierry Coquand. *Une théorie des Constructions.* PhD thesis, Université de Paris VII, 1985.

[5] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76, 1988.

[6] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[7] Christina Cornes and Delphine Terrasse. Automatizing inversion of inductive predicates in coq. In *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[8] Jean-Christophe Filliâtre. Proof of imperative programs in type theory. In *International Workshop TYPES'98*, volume 1657 of *Lecture Notes in Computer Science*. Pringer-Verlag, March 1998.

[9] Jean-Christophe Filliâtre. *Preuve de programmes impératifs en théorie des types.* PhD thesis, Université Paris-Sud, July 1999.

[10] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science : 19th symposium on Applied Mathematics*, pages 19–31, 1966.

[11] Stephen N. Freund and John C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications*, October 1998.

[12] Stephen N. Freund and John C. Mitchell. A Formal Framework for the Java Bytecode Language and Verifier. In *ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, November 1999.

[13] A. Goldberg. A specification of Java loading and bytecode verification. In *Proceedings of 5th ACM Conference on Computer and Communication Security*, 1998.

[14] Charles Anthony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, October 1969.

[15] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[16] Per Martin-Löf. An intuitinistic theory of types. Technical report, University of stockholm, 1972.

[17] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990. ISBN 0-262-63132-6.

[18] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[19] R. O'Callahn. A simple, comprehensive type system for java bytecode subroutines. In *ACM Symposium on Principles of Programming Languages*, pages 70–78. ACM Press, 1999.

[20] David von Oheimb and Tobias Nipkow. Machine checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998. To appear.

[21] Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[22] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[23] Lawrence C. Paulson and Tobias Nipkow. *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[24] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *LNCS*, pages p. 89–103. Springer-Verlag, 1999.

[25] Z. Qian. A formal specification of Java Virtual machine instructions for objects, methods, and subroutines. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.

[26] Joseph Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs, September 1992. (In french), available at URL `http://coq.inria.fr/contribs/unification.html`.

[27] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 149–160. ACM Press, January 1998.

[28] R. Tarjan. On the Efficiency of a Good but not Linear Set Merging Algorithm. In *Journal of the ACM*, volume 22, pages 215–225, 1975.

Contents

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399