CLDC Byte Code Typechecker Specification

Gilad Bracha, Tim Lindholm, Wei Tao and Frank Yellin
Sun Microsystems

January 14, 2003

1. Overview

This document gives a specification of the CLDC *typechecker,* also referred to as the CLDC *runtime verifier*, or KVM *runtime verifier*. The document is intended to supplement the *CLDC Specification* and to provide a detailed description of the runtime verification process characteristic of virtual machines conforming to the *CLDC Specification*.

The type rules that the typechecker enforces are specified by means of Prolog clauses. English language text is used to describe the type rules in an informal way, while the Prolog code provides a formal specification.

The next section describes the format of the type annotations expected by the typechecker. A specification of the typechecker proper is given in §3.

1.1 Notation and Terminology

Whenever this specification refers to a class or interface that is declared in the package `java` or any of its subpackages, the intended reference is to that class or interface as loaded by the bootstrap class loader of the Java Virtual Machine. Whenever we refer to a class or interface using a single identifier ***N***, the intended reference is the class `java.lang.`***N***.

In configurations that do not support an exception or error class that must be thrown according to this specification, the typechecker should do whatever the specification of such configuration prescribes to signal that exception or error.

We use this font for Prolog code and code fragments.

We use `another font` for Java virtual machine instructions and for class file structures.

Commentary, designed to clarify the specification is given as indented text between horizontal lines:

> Commentary provides intuition, motivation, rationale, examples etc.

The type checker deals with and manipulates the expected types of a method's local variables and operand stack. Throughout this document, a *location* refers to either a single local variable or to a single operand stack entry.

We will use the terms *stack frame map* and *type state* interchangeably to describe a mapping between locations in the operand stack and local variables of a method and verification types. We will usually use the term *stack frame map* when such a mapping is provided in the class file, and the term *type state* when the mapping is inferred by the type checker.

2. Parsing

The type checker requires a list of stack frame maps for each method with a `Code` attribute. The type checker reads the stack frame maps for each such method and uses these maps to generate a proof of the type safety of the instructions in the `Code` attribute. The list of stack frame maps is given by the *stack map attribute*. This section specifies the format of the stack map attribute. If the stack map attribute does not conform to the format specified in this section, the Java virtual machine must throw a `java.lang.ClassFormatError`.

> The intent is that a stack frame map must appear at the beginning of each basic block in a method. The stack frame map specifies the verification type of each operand stack entry and of each local variable at the start of each basic block.

2.1 Stack map format

The stack map attribute is an optional variable-length attribute in the attributes table of a `Code` attribute. The name of the attribute is `StackMap`. A stack map attribute consists of zero or more stack map frames. Each stack map frame specifies an offset, an array of verification types for the local variables, and an array of verification types for the operand stack.

If a method's `Code` attribute does not have a `StackMap` attribute, it has an *implicit stack map attribute*. This implicit stack map attribute is equivalent to a `StackMap` attribute with `number_of_entries` equal to zero. A method's `Code` attribute may have at most one `StackMap` attribute, otherwise a `java.lang.ClassFormatError` is thrown.

The format of the stack map in the class file is given below. In the following, if the length of the method's byte code[1] is 65535 or less, then `uoffset` represents the type `u2`; otherwise `uoffset` represents the type `u4`. If the maximum number of local variables for the method is 65535 or less, then `ulocalvar` represents the type `u2`; otherwise `ulocalvar` represents the type `u4`. If the maximum size of the operand stack is 65535 or less, then `ustack` represents the type `u2`; otherwise ustack represents the type `u4`[2].

```
stack_map { // attribute StackMap
      u2 attribute_name_index;
      u4 attribute_length
      uoffset number_of_entries;
      stack_map_frame entries[number_of_entries];
      }
```

Each stack map frame has the following format:

```
stack_map_frame {
      uoffset offset;
      ulocalvar number_of_locals;
      verification_type_info locals[number_of_locals];
      ustack number_of_stack_items;
      verification_type_info stack[number_of_stack_items];
}
```

The 0th entry in `locals` represents the type of local variable 0. If `locals[M]` represents local variable N, then `locals[M+1]` represents local variable N+1 if `locals[M]` is one of `Top_variable_info, Integer_variable_info, Float_variable_info, Null_variable_info, UninitializedThis_variable_info, Object_variable_info,` or

1. Note that the length of the byte codes is not the same as the length of the `Code` attribute. The byte codes are embedded in the `Code` attribute, along with other information.
2. For the J2ME CLDC technology, the size of the fields mentioned in this paragraph is always 16 bits (`u2`).

`Uninitialized_variable_info`, otherwise `locals[M+1]` represents local variable N+2. It is an error if, for any index `i`, `locals[i]` represents a local variable whose index is greater than the maximum number of local variables for the method.

The 0th entry in `stack` represents the type of the bottom of the stack, and subsequent entries represent types of stack elements closer to the top of the operand stack. We shall refer to the bottom element of the stack as stack element 0, and to subsequent elements as stack element 1, 2 etc. If `stack[M]` represents stack element N, then `stack[M+1]` represents stack element N+1 if `stack[M]` is one of `Top_variable_info, Integer_variable_info, Float_variable_info, Null_variable_info, UninitializedThis_variable_info, Object_variable_info,` or `Uninitialized_variable_info`, otherwise `stack[M+1]` represents stack element N+2. It is an error if, for any index `i`, `stack[i]` represents a stack entry whose index is greater than the maximum operand stack size for the method.

We say that an instruction in the byte code has a corresponding stack map frame if the offset in the offset field of the stack map frame is the same as the offset of the instruction in the byte codes.

The `verification_type_info` structure consists of a one-byte tag followed by zero or more bytes, giving more information about the tag. Each `verification_type_info` structure specifies the verification type of one or two locations.

```
union verification_type_info {
      Top_variable_info;
      Integer_variable_info;
      Float_variable_info;
      Long_variable_info;
      Double_variable_info;
      Null_variable_info;
      UninitializedThis_variable_info;
      Object_variable_info;
      Uninitialized_variable_info;
}
```

The `Top_variable_info` indicates that the local variable has the verification type `top` ($\top$.)

```
Top_variable_info {
      u1 tag = ITEM_Top; /* 0 */
}
```

The `Integer_variable_info` type indicates that the location contains the verification type `int`.

```
Integer_variable_info {
      u1 tag = ITEM_Integer; /* 1 */
}
```

The `Float_variable_info` type indicates that the location contains the verification type `float`.

```
Float_variable_info {
      u1 tag = ITEM_Float; /* 2 */
}
```

The `Long_variable_info` type indicates that the location contains the verification type `long`. If the location is a local variable, then:

- It must not be the local variable with the highest index.
- The next higher numbered local variable contains the verification type $\top$.

If the location is an operand stack entry, then:

- The current location must not be the topmost location of the operand stack.
- the next location closer to the top of the operand stack contains the verification type $\top$.

This structure gives the contents of two locations in the `stack[]` or `local[]` array.

```
Long_variable_info {
      u1 tag = ITEM_Long; /* 4 */
}
```

The `Double_variable_info` type indicates that the location contains the verification type `double`. If the location is a local variable, then:

- It must not be the local variable with the highest index.
- The next higher numbered local variable contains the verification type $\top$.

If the location is an operand stack entry, then:

- The current location must not be the topmost location of the operand stack.
- the next location closer to the top of the operand stack contains the verification type $\top$.

This structure gives the contents of two locations in the `stack[]` or `local[]` array.

```
Double_variable_info {
      u1 tag = ITEM_Double; /* 3 */
}
```

The `Null_variable_info` type indicates that location contains the type checker type `null`.

```
Null_variable_info {
      u1 tag = ITEM_Null; /* 5 */
}
```

The `UninitializedThis_variable_info` type indicates that the location contains the verification type `uninitializedThis`.

```
UninitializedThis_variable_info {
      u1 tag = ITEM_UninitializedThis; /* 6 */
}
```

The `Object_variable_info` type indicates that the location contains the class referenced by the constant pool entry.

```
Object_variable_info {
      u1 tag = ITEM_Object; /* 7 */
      u2 cpool_index;
}
```

The `Uninitialized_variable_info` indicates that the location contains the verification type `uninitialized`(*offset*). The `offset` item indicates the offset of the `new` instruction that created the object being stored in the location.

```
Uninitialized_variable_info {
      u1 tag = ITEM_Uninitialized /* 8 */
      uoffset offset;
}
```

3. Typechecking

Typechecking of a class file is performed after the class has been successfully loaded.

Iff the predicate classIsTypeSafe is not true, the type checker must throw the exception `java.lang.VerifyError` to indicate that the class file is malformed. Otherwise, the class file has type checked successfully and byte code verification has completed successfully.

```
classIsTypeSafe(Class) :-
    classClassName(Class,Name),
    Name \= 'java/lang/Object',
    classSuperClassName(Class, SuperclassName),
    loadedClass(SuperclassName, Superclass),
    classIsNotFinal(Superclass),
    classMethods(Class, Methods),
    checklist(methodIsTypeSafe(Class), Methods).

classIsTypeSafe(Class) :-
    classClassName(Class, 'java/lang/Object'),
    classMethods(Class, Methods),
    checklist(methodIsTypeSafe(Class), Methods).
```

The predicate classIsTypeSafe assumes that Class is a Prolog term representing a binary class that has been parsed successfully. This specification does not mandate the precise structure of this term, but does require that certain predicates (e.g., classMethods) be defined upon it, as specified in §3.2.1.

> For example, we assume a predicate classMethods(Class, Methods) that, given a term representing a class as described above as its first argument, binds its second argument to a list comprising all the methods of the class, represented in a convenient form described below.
>
> Thus, a class is type safe if all its methods are type safe, and it does not subclass a final class.

We also require the existence of a predicate loadedClass(Name, ClassDefinition) which asserts that there exists a class named Name, whose representation (in accordance with this specification) is ClassDefinition.

Individual instructions are presented as terms whose functor is the name of the instruction and whose arguments are its parsed operands.

> For example, an `aload` instruction is represented as the term aload(N), which includes the index N that is the operand of the instruction.

A few instructions have operands that are constant pool entries representing methods or fields. As specified in the JVMS, methods are represented by `CONSTANT_InterfaceMethodref_info` (for interface methods) or `CONSTANT_Methodref_info` (for other methods) structures in the constant pool. Such structures are represented here as functor applications of the form imethod(MethodClassName, MethodName, MethodSignature) (for interface methods) or method (MethodClassName, MethodName, MethodDescriptor) (for other methods), where MethodClassName is the name of the class referenced by the `class_index` item for the structure, and

MethodName and MethodDescriptor correspond to the name and type descriptor referenced by the `name_and_type_index` of the structure.

Similarly, fields are represented by `CONSTANT_Fieldref_info` structures in the class file. These structures are represented here as functor applications of the form field(FieldClassName, FieldName, FieldDescriptor) where FieldClassName is the name of the class referenced by the `class_index` item in the structure, and `FieldName` and FieldDescriptor correspond to the name and type descriptor referenced by the `name_and_type_index` item of the structure.

> So, a `getfield` instruction whose operand was an index into the constant pool that refers to a field `foo` of type `F` in class `Bar` would be represented as getfield(field('Bar', 'foo', 'F')).

The instructions as a whole are represented as a list of terms of the form instruction(Offset, AnInstruction).

> For example instruction(21, aload(1)).

The order of instructions in this list must be the same as in the class file.

The `StackMap` attribute is represented as a list of terms of the form stackMap(Offset, TypeState) where Offset is an integer indicating the offset of the instruction the frame map applies to, and TypeState is the expected incoming type state for that instruction. The order of instructions in this list must be the same as in the class file.

TypeState has the form frame(Locals, OperandStack, Flags).

Locals is a list of verification types, such that the *N*th element of the list (with 0 based indexing) represents the type of local variable *N*.

If any local variable in Locals has the type `uninitializedThis`, Flags is `[flagThisUninit]`, otherwise it is an empty list.

OperandStack is a list of types, such that the first element represents the type of the top of the operand stack, and the elements below the top follow in the appropriate order.

However, note again that types of size 2 are represented by two entries, with the first entry being top and the second one being the type itself.

> So a stack with a `double`, an `int` and a `long` would be represented as [top, double, int, top, long].

Array types are represented by applying the functor arrayOf to an argument denoting the element type of the array. Other reference types are represented by applying the functor class to the fully qualified name of the class or interface. The type `uninitialized`(*offset*) is represented by applying the functor uninitialized to an argument representing the numerical value of the *offset*. Other verification types are represented by Prolog atoms whose name denotes the verification type in question.

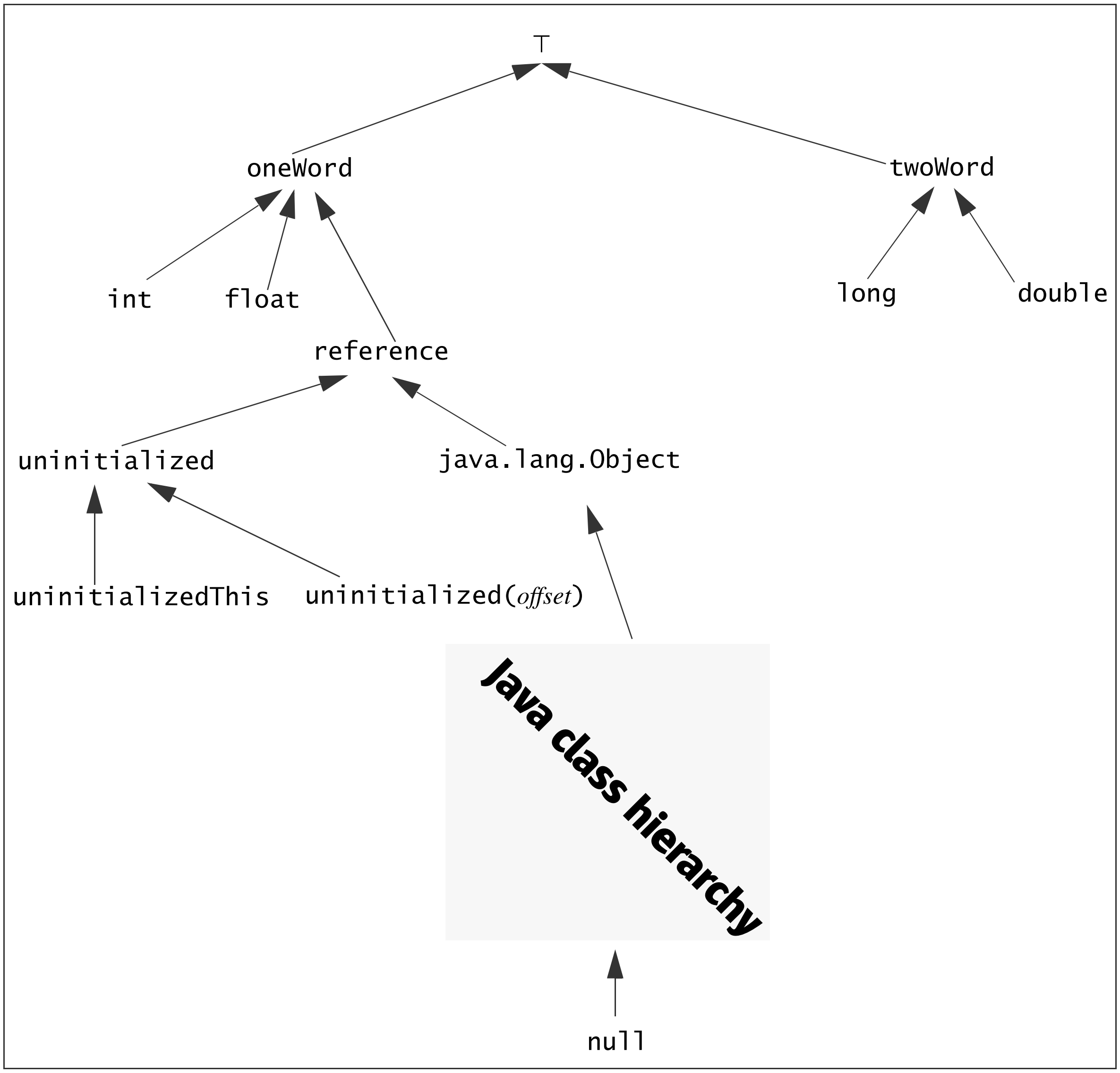
> So, the class `Object` would be represented as class('java/lang/Object'), and the types `int[]` and `Object[]` would be represented by arrayOf(int) and arrayOf(class('java/lang/Object')) respectively.

Flags is a list which may either be empty or have the single element `flagThisUninit`.

> This flag is used in constructors, to mark type states where initialization of `this` has not yet been completed. In such type states, it is illegal to return from the method.

3.1 The Type hierarchy

The typechecker enforces a type system based upon a hierarchy of verification types, illustrated in the figure below. Most verifier types have a direct correspondence with Java virtual machine field type descriptors as given in the JVMS, table 4.2. The only exceptions are the field descriptors `B`, `C`, `S` and `Z` all of which correspond to the verifier type `int`.



The verification type Hierarchy

3.1.1 Subtyping Rules

Subtyping is reflexive

```
isAssignable(X, X) .

isAssignable(oneWord, top).
isAssignable(twoWord, top).

isAssignable(int, X) :- isAssignable(oneWord, X).
isAssignable(float, X) :- isAssignable(oneWord, X).
isAssignable(long, X) :- isAssignable(twoWord, X).
isAssignable(double, X) :- isAssignable(twoWord, X).
isAssignable(reference, X) :- isAssignable(oneWord, X).

isAssignable(uninitialized, X) :- isAssignable(reference, X).
```

These subtype rules are not necessarily the most obvious formulation of subtyping. There is a clear split between subtyping rules for reference types in the Java programming language, and rules for the remaining verification types.

Subtype rules for the reference types in the Java programming language are specified recursively in the obvious way. The remaining verification types have subtypes rules of the form:

```
subtype(v, X) :- subtype(the_direct_supertype_of_v, X).
```

That is, `v` is a subtype of `X` if the direct supertype of `v` is a subtype of `X`.

We also have a rule that says subtyping is reflexive, so together these rules cover most verification types that are not reference types in the Java programming language.

The aforementioned split allows us to state general subtyping relations between the Java programming language types and other verification types.

These relations hold independently of the Java type's position in the hierarchy. These rules are useful for the reference implementation, where they help prevent duplicate answers and excessive class loading.

For example, we don't want to start climbing up the class hierarchy in response to a query of the form `class(foo)` <: `twoWord`. If we use the same predicates for the entire hierarchy, we cannot restrict the climb to cases where we compare two classes.

It would be nicer to have more uniform rules for the specification, but they are not well suited for the reference implementation. We'd like the reference implementation to be as close to the specification as possible, so this is a reasonable compromise.

```
isAssignable(class(_),   X) :- isAssignable(reference, X).
isAssignable(arrayOf(_), X) :- isAssignable(reference, X).

isAssignable(uninitializedThis, X) :- isAssignable(uninitialized, X).
```

```
isAssignable(uninitialized(_),    X) :- isAssignable(uninitialized, X).

isAssignable(null, class(_)).
isAssignable(null, arrayOf(_)).
isAssignable(null, X)  :- isAssignable(class('java/lang/Object'), X).

isAssignable(class(X), class(Y)) :- isJavaAssignable(class(X), class(Y)).
isAssignable(arrayOf(X), class(Y)) :- isJavaAssignable(arrayOf(X), class(Y)).
isAssignable(arrayOf(X), arrayOf(Y)) :- isJavaAssignable(arrayOf(X),arrayOf(Y)).
```

For assignments, interfaces are treated like `Object`.

```
isJavaAssignable(class(_), class(To)) :-
        loadedClass(To, ToClass),
        classIsInterface(ToClass).

isJavaAssignable(class(From), class(To)) :-
        isJavaSubclassOf(From, To).
```

Arrays are subtypes of `Object`.

```
isJavaAssignable(arrayOf(_), class('java/lang/Object')).
```

The intent here is that array types are subtypes of `Cloneable` and `java.io.Serializable` iff the latter two types are defined. In CLDC, these types do not exist. This is reflected in the Prolog code for CLDC by making the predicate `isArrayInterface` have no clauses, and thus always false.

```
isJavaAssignable(arrayOf(_), class(X)) :-
        isArrayInterface(X).
```

The subtyping relation between arrays of primitive type is the identity relation.

```
isJavaAssignable(arrayOf(X), arrayOf(Y)) :-
        atom(X), atom(Y), X = Y.
```

Subtyping between arrays of reference type is covariant.

```
isJavaAssignable(arrayOf(X), arrayOf(Y)) :-
        compound(X), compound(Y), isJavaAssignable(X, Y).
```

Subclassing is reflexive.

```
isJavaSubclassOf(SubClassName, SubClassName).
isJavaSubclassOf(SubClassName, SuperClassName) :-
    loadedClass(SubClassName, SubClass),
    classSuperClassName(SubClass, SubSuperClassName),
    isJavaSubclassOf(SubSuperClassName, SuperClassName).

sizeOf(X, 2) :- isAssignable(X, twoWord).
sizeOf(X, 1) :- isAssignable(X, oneWord).
sizeOf(top, 1).
```

Subtyping is extended pointwise to type states.

```
frameIsAssignable(frame(Locals1, StackMap1, Flags1),
        frame(Locals2, StackMap2, Flags2)) :-
    length(StackMap1, StackMapLength),
    length(StackMap2, StackMapLength),
    maplist(isAssignable, Locals1, Locals2),
    maplist(isAssignable, StackMap1, StackMap2),
    subset(Flags1, Flags2).
```

3.2 Typechecking Rules

3.2.1 Accessors

Stipulated Accessors: Throughout this specification, we assume the existence of certain Prolog predicates whose formal definitions are not given in the specification. In this section, we list these predicates and specify their expected behavior.

```
parseFieldSignature(Signature, Type)
```

Converts a field descriptor, `Signature`, into the corresponding verification type `Type` (see the beginning of §3.1 for the specification of this correspondence).

```
parseMethodSignature(Signature, TypeList, ReturnType)
```

Converts a method descriptor, `Signature`, into a list of verification types, `TypeList`, corresponding (§3.1) to the method argument types, and a verification type, `ReturnType`, corresponding to the return type.

```
parseCodeAttribute(ClassFile, Method, FrameSize, MaxStack, ParsedCode, Handlers, StackMap)
```

Extracts the instruction stream, `ParsedCode`, of the method `Method` in `ClassFile`, as well the maximum operand stack size, `MaxStack`, the maximal number of local variables, `FrameSize`, the exception handlers, `Handlers`, and the stack map `StackMap`. The representation of the instruction stream and stack map attribute must be as specified in the beginning of §3. Each exception handler is represented by a functor application of the form `handler(Start, End, Target, ClassName)` whose arguments are, respectively, the start and end of the range of instructions covered by the handler, the first instruction of the handler code, and the name of the exception class that this handler is designed to handle.

    classClassName(Class, ClassName)

Extracts the name, ClassName, of the class Class.

    classIsInterface(Class, IsInterface)

True iff the class, Class, is an interface.

    classIsNotFinal(Class)

True iff the class, Class, is not a final class.

    classSuperClassName(Class, SuperClassName)

Extracts the name, SuperClassName, of the superclass of class Class.

    classInterfaces(Class, Interfaces)

Extracts a list, Interfaces, of the direct superinterfaces of the class Class.

    classMethods(Class, Methods)

Extracts a list, Methods, of the methods of the class Class.

    classAttributes(Class, Attributes)

Extracts a list, Attributes, of the attributes of the class Class. Each attribute is represented as a functor application of the form attribute(AttributeName, AttributeContents), where AttributeName is the name of the attribute. The format of the attributes contents is unspecified.

    methodName(Method, Name)

Extracts the name, Name, of the method Method.

    methodAccessFlags(Method, AccessFlags)

Extracts the access flags, AccessFlags, of the method Method.

    methodSignature(Method, Signature)

Extracts the descriptor, Signature, of the method Method.

    methodAttributes(Method, Attributes)

Extracts a list, Attributes, of the attributes of the method Method.

    isNotFinal(Method, Class)

True iff Method in class Class is not final.

    isProtected(MemberClassName, MemberName, MemberSignature)

True iff the member named MemberName with signature MemberSignature in the class MemberClassName is protected.

    isNotProtected(MemberClassName, MemberName, MemberSignature)

True iff the member named MemberName with signature MemberSignature in the class MemberClassName is not protected.

> There is a principle guiding the determination as to which accessors are fully specified and which are stipulated. We do not want to over-specify the representation of the class file. Providing specific accessors to the class or method term would force us to completely specify a format for the Prolog term representing the class file.

Specified Accessors and Utilities: We also provide accessor and utility rules that extract necessary information from the representation of the class and its methods.

An environment is a six-tuple consisting of a class, a method, the declared return type of the method, the instructions in a method, the maximal size of the operand stack, and a list of exception handlers.

```
maxOperandStackLength(Environment, MaxStack) :-
    Environment = environment(_Class, _Method, _ReturnType, _All, MaxStack, _Handlers).
exceptionHandlers(Environment, Handlers) :-
    Environment = environment(_Class, _Method, _ReturnType, _All, _, Handlers).

thisMethodReturnType(Environment, ReturnType) :-
    Environment = environment(_Class, _Method, ReturnType, _All, _, _).

thisClass(Environment, class(ClassName)) :-
    Environment = environment(Class, _Method, _ReturnType, _All, _, _),
    classClassName(Class, ClassName).

allInstructions(Environment, All) :-
    Environment = environment(_Class, _Method, _ReturnType, All, _, _).

offsetStackFrame(Environment, Offset, StackFrame) :-
    allInstructions(Environment, Instructions),
    member(stackMap(Offset, StackFrame), Instructions).

notMember(_, []).
notMember(X, [A | More]) :- X \= A, notMember(X, More).
```

3.2.2 Abstract & Native Methods

Abstract methods are considered to be type safe if they do not override a final method.

```
methodIsTypeSafe(Class, Method) :-
    doesNotOverrideFinalMethod(Class, Method),
    methodAccessFlags(Method, AccessFlags),
    member(abstract, AccessFlags).
```

Native methods are considered to be type safe if they do not override a final method.

```
methodIsTypeSafe(Class, Method) :-
        doesNotOverrideFinalMethod(Class, Method),
        methodAccessFlags(Method, AccessFlags),
        member(native, AccessFlags).
doesNotOverrideFinalMethod(class('java/lang/Object'), Method).
doesNotOverrideFinalMethod(Class, Method) :-
        classSuperClassName(Class, SuperclassName),
        loadedClass(SuperclassName, Superclass),
        classMethods(Superclass, MethodList),
        finalMethodNotOverridden(Method, Superclass, MethodList).

finalMethodNotOverridden(Method, Superclass, MethodList) :-
        methodName(Method, Name),
        methodSignature(Method, Sig),
        member(method(_, Name, Sig), MethodList),
        isNotFinal(Method, Superclass).

finalMethodNotOverridden(Method, Superclass, MethodList) :-
        methodName(Method, Name),
        methodSignature(Method, Sig),
        notMember(method(_, Name, Sig), MethodList),
        doesNotOverrideFinalMethod(Superclass, Method).
```

3.2.3 Checking Code

Non-abstract, non-native methods are type correct if they have code and the code is type correct.

```
methodIsTypeSafe(Class, Method) :-
        doesNotOverrideFinalMethod(Class, Method),
        methodAccessFlags(Method, AccessFlags),
        methodAttributes(Method, Attributes),
        notMember(native, AccessFlags),
        notMember(abstract, AccessFlags),
        member(attribute('Code', _), Attributes),
        methodWithCodeIsTypeSafe(Class, Method).
```

A method with code is type safe if it is possible to merge the code and the stack frames into a single stream such that each stack map precedes the instruction it corresponds to, and the resulting merged stream is type correct.

```
methodWithCodeIsTypeSafe(Class, Method) :-
        parseCodeAttribute(Class, Method, FrameSize, MaxStack, ParsedCode, Handlers,
                           StackMap),
        mergeStackMapAndCode(StackMap, ParsedCode, MergedCode),
        methodInitialStackFrame(Class, Method, FrameSize, StackFrame, ReturnType),
        Environment =
            environment(Class, Method, ReturnType, MergedCode, MaxStack, Handlers),
        handlersAreLegal(Environment),
        mergedCodeIsTypeSafe(Environment, MergedCode, StackFrame).
```

The initial type state of a method consists of an empty operand stack and local variable types derived from the type of `this` and the arguments, as well as the appropriate flag, depending on whether this is an `<init>` method.

```
methodInitialStackFrame(Class, Method, FrameSize,
            frame(Locals, [], Flags), ReturnType):-
    methodSignature(Method, Signature),
    parseMethodSignature(Signature, RawArgs, ReturnType),
    expandTypeList(RawArgs, Args),
    methodInitialThisType(Class, Method, ThisList),
    flags(ThisList, Flags),
    append(ThisList, Args, ThisArgs),
    expandToLength(ThisArgs, FrameSize, top, Locals).

flags([uninitializedThis], [flagThisUninit]).
flags(X, []) :- X \= [uninitializedThis].

expandToLength(List, Size, _Filler, List) :- length(List, Size).
expandToLength(List, Size, Filler, Result) :-
    length(List, ListLength),
    ListLength < Size,
    Delta is Size - ListLength,
    length(Extra, Delta),
    checklist(=(Filler), Extra),
    append(List, Extra, Result).
```

For a static method `this` is irrelevant; the list is empty. For an instance method, we get the type of `this` and put it in a list.

```
methodInitialThisType(_Class, Method, []) :-
    methodAccessFlags(Method, AccessFlags),
    member(static, AccessFlags),
    methodName(Method, MethodName),
    MethodName \= '<init>'.
methodInitialThisType(Class, Method, [This]) :-
    methodAccessFlags(Method, AccessFlags),
  notMember(static, AccessFlags),
  instanceMethodInitialThisType(Class, Method, This).
```

In the `<init>` method of Object the type of this is Object. In other `<init>` methods, the type of `this` is `uninitializedThis`. Otherwise, the type of `this` in an instance method is class(N), where N is the name of the class containing the method.

```
instanceMethodInitialThisType(Class, Method, class(ClassName)) :-
    methodName(Method, MethodName),
    MethodName \= '<init>',
    classClassName(Class, ClassName).

instanceMethodInitialThisType(Class, Method, class('java/lang/Object')) :-
    methodName(Method, '<init>'),
    classClassName(Class, 'java/lang/Object').
instanceMethodInitialThisType(Class, Method, uninitializedThis) :-
    methodName(Method, '<init>'),
    classClassName(Class, ClassName),
    ClassName \= 'java/lang/Object'.
```

Below are the rules for iterating through the code stream. The assumption is that the stream is a well formed mixture of instructions and stack maps, such that the stack map for byte code index *N* appears just before instruction *N*.

The rules for building this mixed stream are given later, by the predicate mergeStackMapAndCode.

The special marker `aftergoto` is used to indicate an unconditional branch.

If we have an unconditional branch at the end of the code, stop.

```
mergedCodeIsTypeSafe(_Environment, [endOfCode(Offset)], afterGoto).
```

After an unconditional branch, if we have a stack map giving the type state for the following instructions, we can proceed and typecheck them using the type state provided by the stack map.

```
mergedCodeIsTypeSafe(Environment, [stackMap(Offset, MapFrame) | MoreCode],
                     afterGoto):-
    mergedCodeIsTypeSafe(Environment, MoreCode, MapFrame).
```

If we have a stack map and an incoming type state, the type state must be assignable to the one in the stack map. We may then proceed to type check the rest of the stream with the type state given in the stack map.

```
mergedCodeIsTypeSafe(Environment, [stackMap(Offset, MapFrame) | MoreCode],
            frame(Locals, OperandStack, Flags)) :-
    frameIsAssignable(frame(Locals, OperandStack, Flags), MapFrame),
    mergedCodeIsTypeSafe(Environment, MoreCode, MapFrame).
```

It is illegal to have code after an unconditional branch without a stack frame map being provided for it.

```
mergedCodeIsTypeSafe(_Environment, [instruction(_,_) | _MoreCode], afterGoto) :-
    write_ln('No stack frame after unconditional branch'),
    fail.
```

A merged code stream is type safe relative to an incoming type state T, if it begins with an instruction I that is type safe relative to T, I satisfies its exception handlers, and the tail of the stream is type safe given the type state following that execution of I.

```
mergedCodeIsTypeSafe(Environment, [instruction(Offset,Parse) | MoreCode],
            frame(Locals, OperandStack, Flags)) :-
```

Verify the instruction. NextStackFrame indicates what falls through to the following instruction. ExceptionStackFrame indicates what is passed to exception handlers.

```
    instructionIsTypeSafe(Parse, Environment, Offset, frame(Locals, OperandStack, Flags),
                          NextStackFrame, ExceptionStackFrame),
    instructionSatisfiesHandlers(Environment, Offset, ExceptionStackFrame),
    mergedCodeIsTypeSafe(Environment, MoreCode, NextStackFrame).
```

Branching to a target is type safe if the target has an associated stack frame, Frame, and the current stack frame, StackFrame, is assignable to Frame.

```
targetIsTypeSafe(Environment, StackFrame, Target) :-
    offsetStackFrame(Environment, Target, Frame),
    frameIsAssignable(StackFrame, Frame).
```

3.2.4 Combining Stack Maps and Instruction Streams

The merge of a stream of stack frames and a stream of instructions is defined in this section.

Merging an empty StackMap and a list of instructions yields the original list of instructions.

```
mergeStackMapAndCode([], CodeList, CodeList).
```

Given a list of stack frame maps beginning with the type state for the instruction at Offset, and a list of instructions beginning at Offset, the merged list consists of the head of the stack frame list, followed by the head of the instruction list, followed by the merge of the tails of the two lists.

```
mergeStackMapAndCode([stackMap(Offset, Map) | RestMap],
              [instruction(Offset, Parse) | RestCode],
              [stackMap(Offset, Map),
               instruction(Offset, Parse) | RestMerge]) :-
      mergeStackMapAndCode(RestMap, RestCode, RestMerge).
```

Otherwise, given a list of stack frames beginning with the type state for the instruction at OffsetM, and a list of instructions beginning at OffsetP, then, if OffsetP < OffsetM then the merged list consists of the head of the instruction list, followed by the merge of the stack frame list and the tail of the instruction list.

```
mergeStackMapAndCode([stackMap(OffsetM, Map) | RestMap],
              [instruction(OffsetP, Parse) | RestCode],
              [instruction(OffsetP, Parse) | RestMerge]) :-
      OffsetP < OffsetM,
      mergeStackMapAndCode([stackMap(OffsetM, Map) | RestMap], RestCode,
                                          RestMerge).
```

Otherwise, the merge of the two lists is undefined.

Since the instruction list has monotonically increasing offsets, the merge of the two lists is not defined unless:

- Every stackmap offset has a corresponding instruction offset.
- The stackmaps are in monotonically increasing order.

3.2.5 Exception Handling

An instruction *satisfies its exception handlers* if it satisfies every exception handler that is applicable to the instruction.

```
instructionSatisfiesHandlers(Environment, Offset, ExceptionStackFrame) :-
    exceptionHandlers(Environment, Handlers),
    sublist(isApplicableHandler(Offset), Handlers, ApplicableHandlers),
    checklist(instructionSatisfiesHandler(Environment, ExceptionStackFrame),
                                            ApplicableHandlers).
```

An exception handler is *applicable* to an instruction if the offset of the instruction is greater or equal to the start of the handler's range and less than the end of the handler's range.

```
isApplicableHandler(Offset, handler(Start, End, _Target, _ClassName)) :-
    Offset >= Start,
    Offset < End.
```

An instruction *satisfies* an exception handler if its incoming type state is StackFrame, and the handler's target (the initial instruction of the handler code) is type safe assuming an incoming type state T. The type state T is derived from StackFrame by replacing the operand stack with a stack whose sole element is the handler's exception class.

```
instructionSatisfiesHandler(Environment, StackFrame, Handler) :-
    Handler = handler(_, _, Target, _),
    handlerExceptionClass(Handler, ExceptionClass),
    /* The stack consists of just the exception. */
    StackFrame = frame(Locals, _, Flags),
    targetIsTypeSafe(Environment, frame(Locals, [ ExceptionClass ], Flags),
                     Target).
```

The exception class of a handler is `Throwable` if the handlers class entry is 0, otherwise it is the class named in the handler.

```
handlerExceptionClass(handler(_, _, _, 0), class('java/lang/Throwable')).
handlerExceptionClass(handler(_, _, _, Name), class(Name)) :- Name \= 0.

handlersAreLegal(Environment) :-
    exceptionHandlers(Environment, Handlers),
    checklist(handlerIsLegal(Environment), Handlers).
```

An exception handler is legal if its start (Start) is less than its end (End), there exists an instruction whose offset is equal to Start, there exists an instruction whose offset equals End and the handler's exception class is assignable to the class Throwable.

```
handlerIsLegal(Environment, Handler) :-
    Handler = handler(Start, End, Target, _),
    Start < End,
    allInstructions(Environment, Instructions),
    member(instruction(Start, _), Instructions),
    offsetStackFrame(Environment, Target, _),
    instructionsIncludeEnd(Instructions, End),
    handlerExceptionClass(Handler, ExceptionClass),
    isAssignable(ExceptionClass, class('java/lang/Throwable')).

instructionsIncludeEnd(Instructions, End) :-
    member(instruction(End, _), Instructions).
instructionsIncludeEnd(Instructions, End) :-
    member(endOfCode(End), Instructions).
```

3.3 Instructions

3.3.1 Isomorphic Instructions

Many byte codes have type rules that are completely isomorphic to the rules for other byte codes. If a byte code *b1* is isomorphic to another byte code *b2*, then the type rule for *b1* is the same as the type rule for *b2*.

```
instructionIsTypeSafe(Instruction, Environment, Offset, StackFrame,
                        NextStackFrame, ExceptionStackFrame) :-
    instructionHasEquivalentTypeRule(Instruction, IsomorphicInstruction),
    instructionIsTypeSafe(IsomorphicInstruction, Environment, Offset,
                        StackFrame, NextStackFrame, ExceptionStackFrame).
```

3.3.2 Manipulating the Operand Stack

This section defines the rules for legally manipulating the type state's operand stack. Manipulation of the operand stack is complicated by the fact that some types occupy two entries on the stack. The predicates given in this section take this into account, allowing the rest of the specification to abstract from this issue.

```
canPop(frame(Locals, OperandStack, Flags), Types,
        frame(Locals, PoppedOperandStack, Flags)) :-
    popMatchingList(OperandStack, Types, PoppedOperandStack).

popMatchingList(OperandStack, [], OperandStack).
popMatchingList(OperandStack, [P | Rest], NewOperandStack) :-
    popMatchingType(OperandStack, P, TempOperandStack, _ActualType),
    popMatchingList(TempOperandStack, Rest, NewOperandStack).
```

Pop an individual type off the stack. More precisely, if the logical top of the stack is some subtype of the specified type, `Type`, then pop it. If a type occupies two stack slots, the logical top of stack type is really the type just below the top, and the top of stack is the unusable type `top`.

```
popMatchingType([ActualType | OperandStack], Type, OperandStack, ActualType) :-
    sizeOf(Type, 1),
    isAssignable(ActualType, Type).
popMatchingType([top, ActualType | OperandStack], Type, OperandStack, ActualType) :-
    sizeOf(Type, 2),
    isAssignable(ActualType, Type).
```

Push a logical type onto the stack. The exact behavior varies with the size of the type. If the pushed type is of size 1, we just push it onto the stack. If the pushed type is of size 2, we push it, and then push `top`.

```
pushOperandStack(OperandStack, 'void', OperandStack).
pushOperandStack(OperandStack, Type, [Type | OperandStack]) :-
    sizeOf(Type, 1).
pushOperandStack(OperandStack, Type, [top, Type | OperandStack]) :-
    sizeOf(Type, 2).
```

The length of the operand stack must not exceed the declared maximum stack length.

```
operandStackHasLegalLength(Environment, OperandStack) :-
    length(OperandStack, Length),
    maxOperandStackLength(Environment, MaxStack),
    Length =< MaxStack.
```

Category 1 types, as defined by the JVMS, occupy a single stack slot. Popping a logical type of category 1, `Type`, off the stack is possible if the top of the stack is `Type` and `Type` is not `top` (otherwise it could denote the upper half of a category 2 type). The result is the incoming stack, with the top slot popped off.

```
popCategory1([Type | Rest], Type, Rest) :-
    Type \=top,
    sizeOf(Type, 1).
```

Category 2 types, as defined by the JVMS, occupy two stack slots. Popping a logical type of category 2, `Type`, off the stack is possible if the top of the stack is type `top`, and the slot directly below it is `Type`. The result is the incoming stack, with the top 2 slots popped off.

```
popCategory2([top, Type | Rest], Type, Rest) :-
    sizeOf(Type, 2).

canSafelyPush(Environment, InputOperandStack, Type, OutputOperandStack) :-
    pushOperandStack(InputOperandStack, Type, OutputOperandStack),
    operandStackHasLegalLength(Environment, OutputOperandStack).

canSafelyPushList(Environment, InputOperandStack, Types, OutputOperandStack) :-
    canPushList(InputOperandStack, Types, OutputOperandStack),
    operandStackHasLegalLength(Environment, OutputOperandStack).
```

```
canPushList(InputOperandStack, [Type | Rest], OutputOperandStack) :-
    pushOperandStack(InputOperandStack, Type, InterimOperandStack),
    canPushList(InterimOperandStack, Rest, OutputOperandStack).
canPushList(InputOperandStack, [], InputOperandStack).
```

3.3.3 Loads

All load instructions are variations on a common pattern, varying the type of the value that the instruction loads.

Loading a value of type Type from local variable Index is type safe, if the type of that local variable is ActualType, ActualType is assignable to Type, and pushing ActualType onto the incoming operand stack is a valid type transition that yields a new type state NextStackFrame. After execution of the load instruction, the type state will be NextStackFrame.

```
loadIsTypeSafe(Environment, Index, Type, StackFrame, NextStackFrame) :-
    StackFrame = frame(Locals, _OperandStack, _Flags),
    nth0(Index, Locals, ActualType),
    isAssignable(ActualType, Type),
    validTypeTransition(Environment, [], ActualType, StackFrame, NextStackFrame).
```

3.3.4 Stores

All store instructions are variations on a common pattern, varying the type of the value that the instruction stores.

In general, a store instruction is type safe if the local variable it references is of a type that is a supertype of Type, and the top of the operand stack is of a subtype of Type, where Type is the type the instruction is designed to store.

More precisely, the store is type safe if one can pop a type ActualType that “matches” Type (i.e., is a subtype of Type) off the operand stack, and then legally assign that type the local variable L_{Index}.

```
storeIsTypeSafe(_Environment, Index, Type, frame(Locals, OperandStack, Flags),
    frame(NextLocals, NextOperandStack, Flags)) :-
    popMatchingType(OperandStack, Type, NextOperandStack, ActualType),
    modifyLocalVariable(Index, ActualType, Locals, NextLocals).
```

Given local variables Locals, modifying L_{Index} to have type Type results in the local variable list NewLocals. The modifications are somewhat involved, because some values (and their corresponding types) occupy two local variables. Hence, modifying L_N may require modifying L_{N+1} (because the type will occupy both the N and N+1 slots) or L_{N-1} (because local N used to be the upper half of the two word value/type starting at local N-1, and so local N-1 must be invalidated), or both. This is described further below. We start at L_0 and count up.

```
modifyLocalVariable(Index, Type, Locals, NewLocals) :-
        modifyLocalVariable(0, Index, Type, Locals, NewLocals).
```

Given the suffix of the local variable list starting at index I, LocalsSuffix, modifying local variable Index to have type Type results in the local variable list suffix NewLocalsSuffix.

If I < Index-1, just copy the input to the output and recurse forward. If I = Index-1, the type of local I may change. This can occur if L_I has a type of size 2. Once we set L_{I+1} to the new type (and the corresponding value), the type/value of L_I will be invalidated, as its upper half will be trashed. Then we recurse forward.

When we find the variable, and it only occupies one word, we change it to Type and we're done.

When we find the variable, and it occupies two words, we change its type to Type and the next word to `top`.

```
modifyLocalVariable(I, Index, Type, [Locals1 | LocalsRest],
                [Locals1 | NextLocalsRest] ) :-
        I < Index - 1,
        I1 is I + 1,
        modifyLocalVariable(I1, Index, Type, LocalsRest, NextLocalsRest).

modifyLocalVariable(I, Index, Type, [Locals1 | LocalsRest],
                [NextLocals1 | NextLocalsRest] ) :-
        I =:= Index - 1,
        modifyPreIndexVariable(Locals1, NextLocals1),
        modifyLocalVariable(Index, Index, Type, LocalsRest, NextLocalsRest).

modifyLocalVariable(Index, Index, Type, [_ | LocalsRest],
                [Type | LocalsRest] ) :-
        sizeOf(Type, 1).

modifyLocalVariable(Index, Index, Type, [_, _ | LocalsRest],
              [Type, top | LocalsRest]) :-
        sizeOf(Type, 2).
```

We refer to a local whose index immediately precedes a local whose type will be modified as a *pre-index variable*. The future type of a pre-index variable of type InputType is Result. If the type, Value, of the pre-index local is of size 1, it doesn't change. If the type of the pre-index local, Value, is 2, we need to mark the lower half of its two word value as unusable, by setting its type to `top`.

```
modifyPreIndexVariable(Value, Value) :- sizeOf(Value, 1).
modifyPreIndexVariable(Value,   top) :- sizeOf(Value, 2).
```

Given a list of types, this clause produces a list where every type of size 2 has been substituted by two entries: one for itself, and one `top` entry. The result then corresponds to the representation of the list as 32 bit words in the Java virtual machine.

```
expandTypeList([], []).

expandTypeList([Item | List], [Item | Result]) :-
    sizeOf(Item, 1),
    expandTypeList(List, Result).

expandTypeList([Item | List], [Item, top | Result]) :-
    sizeOf(Item, 2),
    expandTypeList(List, Result).
```

3.3.5 List of all Instructions

In general, the type rule for an instruction is given relative to an environment Environment that defines the class and method in which the instruction occurs, and the offset Offset within the method at which the instruction occurs. The rule states that if the incoming type state StackFrame fulfills certain requirements, then

- The instruction is type safe.
- It is provable that the type state after the instruction completes normally has a particular form given by NextStackFrame, and that the type state after the instruction completes abruptly is given by ExceptionStackFrame.

The natural language description of the rule is intended to be readable, intuitive and concise. As such, the description avoids repeating all the contextual assumptions given above. In particular:

- We do not explicitly mention the environment.
- When we speak of the operand stack or local variables in the following, we are referring to the operand stack and local variable components of a type state: either the incoming type state or the outgoing one.
- The type state after the instruction completes abruptly is almost always identical to the incoming type state. We only discuss the type state after the instruction completes abruptly when that is not the case.
- We speak of popping and pushing types onto the operand stack. We do not explicitly discuss issues of stack underflow or overflow, but assume that these operations can be completed successfully. The formal rules for operand stack manipulation ensure that the necessary checks are made.
- Similarly, the text discusses only the manipulation of logical types. In practice, some types take more than one word. We abstract from these representation details in our discussion, but the logical rules that manipulate data do not.

Any ambiguities can be resolved by referring to the formal Prolog rules.

aaload:

An `aaload` instruction is type safe iff one can validly replace types matching `int` and an array type with element type ElementType where ElementType is a subtype of `Object`, with ElementType yielding the outgoing type state.

```
instructionIsTypeSafe(aaload, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
      nth1OperandStackIs(2, StackFrame, ArrayType),
      arrayElementType(ArrayType, ElementType),
      validTypeTransition(Environment,
                       [int, arrayOf(class('java/lang/Object'))], ElementType,
                       StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The element type of an array of X is X.

```
arrayElementType(arrayOf(X), X).
```

We define the element type of `null` to be `null`.

```
arrayElementType(null, null).
```

aastore:

An `aastore` instruction is type safe iff one can validly pop types matching `Object`, `int`, and an array of `Object` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(aastore, _Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
      canPop(StackFrame, [class('java/lang/Object'), int,
      arrayOf(class('java/lang/Object'))], NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

aconst_null:

An `aconst_null` instruction is type safe if one can validly push the type `null` onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(aconst_null, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
      validTypeTransition(Environment, [], null, StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

aload:

An `aload` instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a load instruction with operand Index and type `reference` is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(aload(Index), Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, reference, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

aload_<n>:

The instructions `aload_<n>`, for $0 \le n \le 3$, are typesafe iff the equivalent `aload` instruction is type safe.

```
instructionHasEquivalentTypeRule(aload_0, aload(0)).
instructionHasEquivalentTypeRule(aload_1, aload(1)).
instructionHasEquivalentTypeRule(aload_2, aload(2)).
instructionHasEquivalentTypeRule(aload_3, aload(3)).
```

anewarray:

An `anewarray` instruction with operand CP is type safe iff CP refers to a constant pool entry denoting either a class type or an array type, and one can legally replace a type matching `int` on the incoming operand stack with an array with component type CP yielding the outgoing type state.

```
instructionIsTypeSafe(anewarray(CP), Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    (CP = class(_) ; CP = arrayOf(_)),
    validTypeTransition(Environment, [int], arrayOf(CP),
                    StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

areturn:

An `areturn` instruction is type safe iff the enclosing method has a declared return type, ReturnType, that is a reference type, and one can validly pop a type matching ReturnType off the incoming operand stack.

```
instructionIsTypeSafe(areturn, Environment, _Offset, StackFrame,
              afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, ReturnType),
    isAssignable(ReturnType, reference),
    canPop(StackFrame, [ReturnType], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

arraylength:

An `arraylength` instruction is type safe iff one can validly replace an array type on the incoming operand stack with the type `int` yielding the outgoing type state.

```
instructionIsTypeSafe(arraylength, Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
      nth1OperandStackIs(1, StackFrame, ArrayType),
      arrayElementType(ArrayType, _),   % ensure that it is an Array
      validTypeTransition(Environment, [top], int, StackFrame, NextStackFrame),
                          exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

astore:

An `astore` instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a store instruction with operand Index and type `reference` is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(astore(Index), Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
      storeIsTypeSafe(Environment, Index, reference, StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

astore_<n>:

The instructions `astore_<n>`, for $0 \le n \le 3$, are typesafe iff the equivalent `astore` instruction is type safe.

```
instructionHasEquivalentTypeRule(astore_0, astore(0)).
instructionHasEquivalentTypeRule(astore_1, astore(1)).
instructionHasEquivalentTypeRule(astore_2, astore(2)).
instructionHasEquivalentTypeRule(astore_3, astore(3)).
```

athrow:

An `athrow` instruction is type safe iff the top of the operand stack matches `Throwable`.

```
instructionIsTypeSafe(athrow, _Environment, _Offset, StackFrame,
              afterGoto, ExceptionStackFrame) :-
      canPop(StackFrame, [class('java/lang/Throwable')], _PoppedStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

baload:

A `baload` instruction is type safe iff one can validly replace types matching `int` and a small array type on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(baload, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
      nth1OperandStackIs(2, StackFrame, Array),
       isSmallArray(Array),
      validTypeTransition(Environment, [int, top], int,
                          StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

An array type is a *small array type* if it is an array of `byte`, an array of `boolean`, or a subtype thereof (`null`).

```
isSmallArray(arrayOf(byte)).
isSmallArray(arrayOf(boolean)).
isSmallArray(null).
```

bastore:

A `bastore` instruction is type safe iff one can validly pop types matching `int, int` and a small array type off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(bastore, _Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
      nth1OperandStackIs(3, StackFrame, Array),
      isSmallArray(Array),
      canPop(StackFrame, [int, int, top], NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

bipush:

A `bipush` instruction is type safe iff the equivalent `sipush` instruction is type safe

```
instructionHasEquivalentTypeRule(bipush(Value), sipush(Value)).
```

caload:

A `caload` instruction is type safe iff one can validly replace types matching `int` and array of `char` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(caload, Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
      validTypeTransition(Environment, [int, arrayOf(char)], int,
                          StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

castore:

A `castore` instruction is type safe iff one can validly pop types matching `int, int` and array of `char` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(castore, _Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
      canPop(StackFrame, [int, int, arrayOf(char)], NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

checkcast:

A `checkcast` instruction with operand CP is type safe iff CP refers to a constant pool entry denoting either a class or an array, and one can validly replace the type `Object` on top of the incoming operand stack with the type denoted by CP yielding the outgoing type state.

```
instructionIsTypeSafe(checkcast(CP), Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
      (CP = class(_) ; CP = arrayOf(_)),
      validTypeTransition(Environment, [class('java/lang/Object')], CP,
                          StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

d2f:

A `d2f` instruction is type safe if one can validly pop `double` off the incoming operand stack and replace it with `float`, yielding the outgoing type state.

```
instructionIsTypeSafe(d2f, Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
      validTypeTransition(Environment, [double], float,
                          StackFrame, NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

d2i:

A `d2i` instruction is type safe if one can validly pop `double` off the incoming operand stack and replace it with `int`, yielding the outgoing type state.

```
instructionIsTypeSafe(d2i, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [double], int,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

d2l:

A `d2l` instruction is type safe if one can validly pop `double` off the incoming operand stack and replace it with `long`, yielding the outgoing type state.

```
instructionIsTypeSafe(d2l, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [double], long,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dadd:

A `dadd` instruction is type safe iff one can validly replace types matching `double` and `double` on the incoming operand stack with `double` yielding the outgoing type state.

```
instructionIsTypeSafe(dadd, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
  validTypeTransition(Environment, [double, double], double,
                      StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

daload:

A `daload` instruction is type safe iff one can validly replace types matching `int` and array of `double` on the incoming operand stack with `double` yielding the outgoing type state.

```
instructionIsTypeSafe(daload, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int, arrayOf(double)], double,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dastore:

A `dastore` instruction is type safe iff one can validly pop types matching `double, int` and array of `double` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(dastore, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [double, int, arrayOf(double)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dcmp<op>:

A `dcmpg` instruction is type safe iff one can validly replace types matching `double` and `double` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(dcmpg, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [double, double], int,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A `dcmpl` instruction is type safe iff the equivalent `dcmpg` instruction is type safe.

```
instructionHasEquivalentTypeRule(dcmpl, dcmpg).
```

dconst_<d>:

A `dconst_0` instruction is type safe if one can validly push the type `double` onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(dconst_0, Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A `dconst_1` instruction is type safe iff the equivalent `dconst_0` instruction is type safe.

```
instructionHasEquivalentTypeRule(dconst_1, dconst_0).
```

ddiv:

A `ddiv` instruction is type safe iff the equivalent `dadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(ddiv, dadd).
```

dload:

A `dload` instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a load instruction with operand Index and type `double` is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(dload(Index), Environment, _Offset, StackFrame,
                  NextStackFrame, ExceptionStackFrame) :-
        loadIsTypeSafe(Environment, Index, double, StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dload_<n>:

The instructions `dload_<n>`, for $0 \le n \le 3$, are typesafe iff the equivalent `dload` instruction is type safe.

```
instructionHasEquivalentTypeRule(dload_0, dload(0)).
instructionHasEquivalentTypeRule(dload_1, dload(1)).
instructionHasEquivalentTypeRule(dload_2, dload(2)).
instructionHasEquivalentTypeRule(dload_3, dload(3)).
```

dmul:

A `dmul` instruction is type safe iff the equivalent `dadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(dmul, dadd).
```

dneg:

A `dneg` instruction is type safe iff there is a type matching `double` on the incoming operand stack. The `dneg` instruction does not alter the type state.

```
instructionIsTypeSafe(dneg, Environment, _Offset, StackFrame,
                  NextStackFrame, ExceptionStackFrame) :-
        validTypeTransition(Environment, [double], double, StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

drem:

A `drem` instruction is type safe iff the equivalent `dadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(drem, dadd).
```

dreturn:

A `dreturn` instruction is type safe if the enclosing method has a declared return type of `double`, and one can validly pop a type matching `double` off the incoming operand stack.

```
instructionIsTypeSafe(dreturn, Environment, _Offset, StackFrame,
                afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, double),
    canPop(StackFrame, [double], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dstore:

A `dstore` instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a store instruction with operand Index and type `double` is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(dstore(Index), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dstore_<n>:

The instructions `dstore_<n>`, for $0 \le n \le 3$, are type safe iff the equivalent `dstore` instruction is type safe.

```
instructionHasEquivalentTypeRule(dstore_0, dstore(0)).
instructionHasEquivalentTypeRule(dstore_1, dstore(1)).
instructionHasEquivalentTypeRule(dstore_2, dstore(2)).
instructionHasEquivalentTypeRule(dstore_3, dstore(3)).
```

dsub:

A `dsub` instruction is type safe iff the equivalent `dadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(dsub, dadd).
```

dup:

A dup instruction is type safe iff one can validly replace a category 1 type, Type, with the types Type, Type, yielding the outgoing type state.

```
instructionIsTypeSafe(dup, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    popCategory1(InputOperandStack, Type, _),
    canSafelyPush(Environment, InputOperandStack, Type, OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dup_x1:

A dup_x1 instruction is type safe iff one can validly replace two category 1 types, Type1, and Type2, on the incoming operand stack with the types Type1, Type2, Type1, yielding the outgoing type state.

```
instructionIsTypeSafe(dup_x1, Environment, _Offset, StackFrame,
               NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type2, Type1], OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

dup_x2:

A dup_x2 instruction is type safe iff it is a type safe form of the dup_x2 instruction.

```
instructionIsTypeSafe(dup_x2, Environment, _Offset, StackFrame,
               NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    dup_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A dup_x2 instruction is *a type safe form of the* dup_x2 *instruction* iff it is a type safe form 1 dup_x2 instruction or a type safe form 2 dup_x2 instruction.

```
dup_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).

dup_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
```

A dup_x2 instruction is *a type safe form 1* dup_x2 *instruction* iff one can validly replace three category 1 types, Type1, Type2, Type3 on the incoming operand stack with the types Type1, Type3, Type2, Type1, yielding the outgoing type state.

```
dup_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Stack2),
    popCategory1(Stack2, Type3, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type3, Type2, Type1],
                      OutputOperandStack).
```

A dup_x2 instruction is *a type safe form 2* dup_x2 *instruction* iff one can validly replace a category 1 type, Type1, and a category 2 type, Type2, on the incoming operand stack with the types Type1, Type2, Type1, yielding the outgoing type state.

```
dup_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory2(Stack1, Type2, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type2, Type1], OutputOperandStack).
```

dup2:

A dup2 instruction is type safe iff it is a type safe form of the dup2 instruction.

```
instructionIsTypeSafe(dup2, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    dup2SomeFormIsTypeSafe(Environment,InputOperandStack, OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A dup2 instruction is *a type safe form of the* dup2 *instruction* iff it is a type safe form 1 dup2 instruction or a type safe form 2 dup2 instruction.

```
dup2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
        dup2Form1IsTypeSafe(Environment,InputOperandStack, OutputOperandStack).
dup2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
        dup2Form2IsTypeSafe(Environment,InputOperandStack, OutputOperandStack).
```

A dup2 instruction is *a type safe form 1* dup2 *instruction* iff one can validly replace two category 1 types, Type1, Type2, on the incoming operand stack with the types Type2, Type1, yielding the outgoing type state.

```
dup2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack):-
        popCategory1(InputOperandStack, Type1, Stack1),
        popCategory1(Stack1, Type2, _),
        canSafelyPushList(Environment, InputOperandStack, [Type2, Type1],
                              OutputOperandStack).
```

A dup2 instruction is *a type safe form 2* dup2 *instruction* iff one can validly replace a category 2 type, Type on the incoming operand stack with the types Type, Type, yielding the outgoing type state.

```
dup2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack):-
        popCategory2(InputOperandStack, Type, _),
        canSafelyPush(Environment, InputOperandStack, Type, OutputOperandStack).
```

dup2_x1:

A dup2_x1 instruction is type safe iff it is a type safe form of the dup2_x1 instruction.

```
instructionIsTypeSafe(dup2_x1, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        StackFrame = frame(Locals, InputOperandStack, Flags),
        dup2_x1SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack),
        NextStackFrame = frame(Locals, OutputOperandStack, Flags),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A dup2_x1 instruction is *a type safe form of the* dup2_x1 *instruction* iff it is a type safe form 1 dup2_x1 instruction or a type safe form 2 dup_x2 instruction.

```
dup2_x1SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
        dup2_x1Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
dup2_x1SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
        dup2_x1Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
```

A dup2_x1 instruction is *a type safe form 1* dup2_x1 *instruction* iff one can validly replace three category 1 types, Type1, Type2, Type3, on the incoming operand stack with the types Type2, Type1, Type3, Type2, Type1, yielding the outgoing type state.

```
dup2_x1Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Stack2),
    popCategory1(Stack2, Type3, Rest),
    canSafelyPushList(Environment, Rest,
                      [Type2, Type1, Type3, Type2, Type1], OutputOperandStack).
```

A dup2_x1 instruction is *a type safe form 2* dup2_x1 *instruction* iff one can validly replace a category 2type, Type1, and a category 1type, Type2, on the incoming operand stack with the types Type1, Type2, Type1, yielding the outgoing type state.

```
dup2_x1Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory2(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type2, Type1], OutputOperandStack).
```

dup2_x2:

A dup2_x2 instruction is type safe iff it is a type safe form of the dup2_x2 instruction.

```
instructionIsTypeSafe(dup2_x2, Environment, _Offset, StackFrame,
            NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    dup2_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A dup2_x2 instruction is *a type safe form of the* dup2_x2 *instruction* iff one of the following holds:

- it is a type safe form 1 dup2_x2 instruction.
- it is a type safe form 2 dup_x2 instruction.
- it is a type safe form 3 dup_x2 instruction.
- it is a type safe form 4 dup_x2 instruction.

```
dup2_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
dup2_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
dup2_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x2Form3IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
dup2_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup2_x2Form4IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
```

A dup2_x2 instruction is *a type safe form 1* dup2_x2 *instruction* iff one can validly replace four category 1 types, Type1, Type2, Type3, Type4, on the incoming operand stack with the types Type2, Type1, Type4, Type3, Type2, Type1, yielding the outgoing type state.

```
dup2_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Stack2),
    popCategory1(Stack2, Type3, Stack3),
    popCategory1(Stack3, Type4, Rest),
    canSafelyPushList(Environment, Rest,
                      [Type2, Type1, Type4, Type3, Type2, Type1], OutputOperandStack).
```

A dup2_x2 instruction is *a type safe form 2* dup2_x2 *instruction* iff one can validly replace a category 2 type, Type1, and two category 1 types, Type2, Type3, on the incoming operand stack with the types Type1, Type3, Type2, Type1, yielding the outgoing type state.

```
dup2_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory2(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Stack2),
    popCategory1(Stack2, Type3, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type3, Type2, Type1], OutputOperandStack).
```

A dup2_x2 instruction is *a type safe form 3* dup2_x2 *instruction* iff one can validly replace two category 1 types, Type1, Type2, and a category 2 type, Type3, on the incoming operand stack with the types Type2, Type1, Type3, Type2, Type1, yielding the outgoing type state.

```
dup2_x2Form3IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Stack2),
    popCategory2(Stack2, Type3, Rest),
    canSafelyPushList(Environment, Rest, [Type2, Type1, Type3, Type2, Type1],
                      OutputOperandStack).
```

A dup2_x2 instruction is *a type safe form 4* dup2_x2 *instruction* iff one can validly replace two category 2 types, Type1, Type2, on the incoming operand stack with the types Type1, Type2, Type1, yielding the outgoing type state.

```
dup2_x2Form4IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory2(InputOperandStack, Type1, Stack1),
    popCategory2(Stack1, Type2, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type2, Type1], OutputOperandStack).
```

f2d:

An f2d instruction is type safe if one can validly pop float off the incoming operand stack and replace it with double, yielding the outgoing type state.

```
instructionIsTypeSafe(f2d, Environment, _Offset, StackFrame,
        NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [float], double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

f2i:

An f2i instruction is type safe if one can validly pop float off the incoming operand stack and replace it with int, yielding the outgoing type state.

```
instructionIsTypeSafe(f2i, Environment, _Offset, StackFrame,
        NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [float], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

f2l:

An `f2l` instruction is type safe if one can validly pop `float` off the incoming operand stack and replace it with `long`, yielding the outgoing type state.

```
instructionIsTypeSafe(f2l, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [float], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fadd:

An `fadd` instruction is type safe iff one can validly replace types matching `float` and `float` on the incoming operand stack with `float` yielding the outgoing type state.

```
instructionIsTypeSafe(fadd, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [float, float], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

faload:

An `faload` instruction is type safe iff one can validly replace types matching `int` and array of `float` on the incoming operand stack with `float` yielding the outgoing type state.

```
instructionIsTypeSafe(faload, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int, arrayOf(float)], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fastore:

An `fastore` instruction is type safe iff one can validly pop types matching `float, int` and array of `float` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(fastore, _Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [float, int, arrayOf(float)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fcmp<op>:

An `fcmpg` instruction is type safe iff one can validly replace types matching `float` and `float` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(fcmpg, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [float, float], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

An `fcmpl` instruction is type safe iff the equivalent `fcmpg` instruction is type safe.

```
instructionHasEquivalentTypeRule(fcmpl, fcmpg).
```

fconst_<f>:

An `fconst_0` instruction is type safe if one can validly push the type `float` onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(fconst_0, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The rules for the other variants of `fconst` are equivalent:

```
instructionHasEquivalentTypeRule(fconst_1, fconst_0).
instructionHasEquivalentTypeRule(fconst_2, fconst_0).
```

fdiv:

An `fdiv` instruction is type safe iff the equivalent `fadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(fdiv, fadd).
```

fload:

An `fload` instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a load instruction with operand Index and type `float` is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(fload(Index), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fload_<n>:

The instructions `fload_<n>`, for $0 \le n \le 3$, are typesafe iff the equivalent `fload` instruction is type safe.

```
instructionHasEquivalentTypeRule(fload_0, fload(0)).
instructionHasEquivalentTypeRule(fload_1, fload(1)).
instructionHasEquivalentTypeRule(fload_2, fload(2)).
instructionHasEquivalentTypeRule(fload_3, fload(3)).
```

fmul:

An `fmul` instruction is type safe iff the equivalent `fadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(fmul, fadd).
```

fneg:

An `fneg` instruction is type safe iff there is a type matching `float` on the incoming operand stack. The `fneg` instruction does not alter the type state.

```
instructionIsTypeSafe(fneg, Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [float], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

frem:

An `frem` instruction is type safe iff the equivalent `fadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(frem, fadd).
```

freturn:

An `freturn` instruction is type safe if the enclosing method has a declared return type of `float`, and one can validly pop a type matching `float` off the incoming operand stack.

```
instructionIsTypeSafe(freturn, Environment, _Offset, StackFrame,
              afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, float),
    canPop(StackFrame, [float], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fstore:

An `fstore` instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a store instruction with operand Index and type `float` is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(fstore(Index), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

fstore_<n>:

The instructions `fstore_<n>`, for $0 \leq n \leq 3$, are typesafe iff the equivalent `fstore` instruction is type safe.

```
instructionHasEquivalentTypeRule(fstore_0, fstore(0)).
instructionHasEquivalentTypeRule(fstore_1, fstore(1)).
instructionHasEquivalentTypeRule(fstore_2, fstore(2)).
instructionHasEquivalentTypeRule(fstore_3, fstore(3)).
```

fsub:

An `fsub` instruction is type safe iff the equivalent `fadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(fsub, fadd).
```

getfield:

A `getfield` instruction with operand CP is type safe iff CP refers to a constant pool entry denoting a field whose declared type is FieldType, declared in a class FieldClass, and one can validly replace a type matching FieldClass with type FieldType on the incoming operand stack yielding the outgoing type state. Protected fields are subject to additional checks.

```
instructionIsTypeSafe(getfield(CP), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    CP = field(FieldClass, FieldName, FieldSignature),
    parseFieldSignature(FieldSignature, FieldType),
    passesProtectedCheck(Environment, FieldClass, FieldName, FieldSignature,
                        StackFrame),
    validTypeTransition(Environment, [class(FieldClass)], FieldType,
                        StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The protected check applies only to members of superclasses of the current class. Other cases will be caught by the access checking done at resolution time.

```
passesProtectedCheck(Environment, MemberClassName, MemberName, MemberSignature,
                     StackFrame) :-
    thisClass(Environment, class(CurrentClassName)),
    superclassChain(CurrentClassName, Chain),
    notMemberOf(MemberClassName, Chain).
```

Using a superclass member that is not protected is trivially correct.

```
passesProtectedCheck(Environment, MemberClassName, MemberName, MemberSignature,
                     StackFrame) :-
    thisClass(Environment, class(CurrentClassName)),
    superclassChain(CurrentClassName, Chain),
    member(MemberClassName, Chain),
    isNotProtected(MemberClassName, MemberName, MemberSignature).
```

Use of a protected superclass member of an object of type Target requires that Target be assignable to the type of the current class.

```
passesProtectedCheck(Environment, MemberClassName, MemberName, MemberSignature,
                     [Target, _Rest]) :-
    thisClass(Environment, class(CurrentClassName)),
    superclassChain(CurrentClassName, Chain),
    member(MemberClassName, Chain),
    isProtected(MemberClassName, MemberName, MemberSignature),
    isAssignable(Target, class(CurrentClassName)).

superclassChain(ClassName, [SuperclassName | Rest]) :-
    classSuperclassName(class(ClassName), SuperclassName),
    superclassChain(SuperclassName, Rest).
superclassChain('java/lang/Object', []).
```

getstatic:

A `getstatic` instruction with operand CP is type safe iff CP refers to a constant pool entry denoting a field whose declared type is FieldType, and one can validly push FieldType on the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(getstatic(CP), Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    CP = field(_FieldClass, _FieldName, FieldSignature),
    parseFieldSignature(FieldSignature, FieldType),
    validTypeTransition(Environment, [], FieldType, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

goto:

A `goto` instruction is type safe iff its target operand is a valid branch target.

```
instructionIsTypeSafe(goto(Target), Environment, _Offset, StackFrame,
              afterGoto, ExceptionStackFrame) :-
    targetIsTypeSafe(Environment, StackFrame, Target),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

i2b:

An `i2b` instruction is type safe iff the equivalent `ineg` instruction is type safe.

```
instructionHasEquivalentTypeRule(i2b, ineg).
```

i2c:

An `i2c` instruction is type safe iff the equivalent `ineg` instruction is type safe.

```
instructionHasEquivalentTypeRule(i2c, ineg).
```

i2d:

An `i2d` instruction is type safe if one can validly pop `int` off the incoming operand stack and replace it with `double`, yielding the outgoing type state.

```
instructionIsTypeSafe(i2d, Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int], double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

i2f:

An `i2f` instruction is type safe if one can validly pop `int` off the incoming operand stack and replace it with `float`, yielding the outgoing type state.

```
instructionIsTypeSafe(i2f, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

i2l:

An `i2l` instruction is type safe if one can validly pop `int` off the incoming operand stack and replace it with `long`, yielding the outgoing type state.

```
instructionIsTypeSafe(i2l, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

i2s:

An `i2s` instruction is type safe iff the equivalent `ineg` instruction is type safe.

```
instructionHasEquivalentTypeRule(i2s, ineg).
```

iadd:

An `iadd` instruction is type safe iff one can validly replace types matching `int` and `int` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(iadd, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int, int], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

iaload:

An `iaload` instruction is type safe iff one can validly replace types matching `int` and array of `int` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(iaload, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int, arrayOf(int)], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

iand:

An `iand` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(iand, iadd).
```

iastore:

An `iastore` instruction is type safe iff one can validly pop types matching `int, int` and array of `int` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(iastore, _Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        canPop(StackFrame, [int, int, arrayOf(int)], NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

if_acmp<cond>:

An `if_acmpeq` instruction is type safe iff one can validly pop types matching `reference` and `reference` on the incoming operand stack yielding the outgoing type state NextStackFrame, and the operand of the instruction, Target, is a valid branch target assuming an incoming type state of NextStackFrame.

```
instructionIsTypeSafe(if_acmpeq(Target), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        canPop(StackFrame, [reference, reference], NextStackFrame),
        targetIsTypeSafe(Environment, NextStackFrame, Target),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The rule for `if_acmp_ne` is identical.

```
instructionHasEquivalentTypeRule(if_acmpne(Target), if_acmpeq(Target)).
```

if_icmp<cond>:

An `if_icmpeq` instruction is type safe iff one can validly pop types matching `int` and `int` on the incoming operand stack yielding the outgoing type state NextStackFrame, and the operand of the instruction, Target, is a valid branch target assuming an incoming type state of NextStackFrame.

```
instructionIsTypeSafe(if_icmpeq(Target), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        canPop(StackFrame, [int, int], NextStackFrame),
        targetIsTypeSafe(Environment, NextStackFrame, Target),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The rules for all other variants of the `if_icmp` instruction are identical

```
instructionHasEquivalentTypeRule(if_icmpge(Target), if_icmpeq(Target)).
instructionHasEquivalentTypeRule(if_icmpgt(Target), if_icmpeq(Target)).
instructionHasEquivalentTypeRule(if_icmple(Target), if_icmpeq(Target)).
instructionHasEquivalentTypeRule(if_icmplt(Target), if_icmpeq(Target)).
instructionHasEquivalentTypeRule(if_icmpne(Target), if_icmpeq(Target)).
```

if_<cond>:

An `if_eq` instruction is type safe iff one can validly pop a type matching `int` off the incoming operand stack yielding the outgoing type state NextStackFrame, and the operand of the instruction, Target, is a valid branch target assuming an incoming type state of NextStackFrame.

```
instructionIsTypeSafe(ifeq(Target), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        canPop(StackFrame, [int], NextStackFrame),
        targetIsTypeSafe(Environment, NextStackFrame, Target),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The rules for all other variations of the `if<cond>` instruction are identical

```
instructionHasEquivalentTypeRule(ifge(Target), ifeq(Target)).
instructionHasEquivalentTypeRule(ifgt(Target), ifeq(Target)).
instructionHasEquivalentTypeRule(ifle(Target), ifeq(Target)).
instructionHasEquivalentTypeRule(iflt(Target), ifeq(Target)).
instructionHasEquivalentTypeRule(ifne(Target), ifeq(Target)).
```

ifnonnull:

An `ifnonnull` instruction is type safe iff one can validly pop a type matching `reference` off the incoming operand stack yielding the outgoing type state NextStackFrame, and the operand of the instruction, Target, is a valid branch target assuming an incoming type state of NextStackFrame.

```
instructionIsTypeSafe(ifnonnull(Target), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        canPop(StackFrame, [reference], NextStackFrame),
        targetIsTypeSafe(Environment, NextStackFrame, Target),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ifnull:

An `ifnull` instruction is type safe iff the equivalent `ifnonnull` instruction is type safe.

```
instructionHasEquivalentTypeRule(ifnull(Target), ifnonnull(Target)).
```

iinc:

An `iinc` instruction with first operand Index is type safe iff L_{Index} has type `int`. The `iinc` instruction does not change the type state.

```
instructionIsTypeSafe(iinc(Index, _Value), _Environment, _Offset, StackFrame,
                StackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, _OperandStack, _Flags),
    nth0(Index, Locals, int),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

iload:

An `iload` instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a load instruction with operand Index and type `int` is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(iload(Index), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

iload_<n>:

The instructions `iload_<n>`, for $0 \le n \le 3$, are typesafe iff the equivalent `iload` instruction is type safe.

```
instructionHasEquivalentTypeRule(iload_0, iload(0)).
instructionHasEquivalentTypeRule(iload_1, iload(1)).
instructionHasEquivalentTypeRule(iload_2, iload(2)).
instructionHasEquivalentTypeRule(iload_3, iload(3)).
```

imul:

An `imul` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(imul, iadd).
```

ineg:

An `ineg` instruction is type safe iff there is a type matching `int` on the incoming operand stack. The `ineg` instruction does not alter the type state.

```
instructionIsTypeSafe(ineg, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        validTypeTransition(Environment, [int], int, StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

instanceof:

An `instanceof` instruction with operand CP is type safe iff CP refers to a constant pool entry denoting either a class or an array, and one can validly replace the type `Object` on top of the incoming operand stack with type `int` yielding the outgoing type state.

```
instructionIsTypeSafe(instanceof(CP), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        (CP = class(_) ; CP = arrayOf(_)),
        validTypeTransition(Environment, [class('java/lang/Object')], int, StackFrame,
                        NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

invokeinterface:

An `invokeinterface` instruction is type safe iff all of the following conditions hold:

- Its first operand, CP, refers to a constant pool entry denoting an interface method named MethodName with signature Signature that is a member of an interface MethodClassName.
- MethodName is not `<init>`.
- MethodName is not `<clinit>`.
- Its second operand, Count, is a valid count operand (see below).
- One can validly replace types matching the type MethodClassName and the argument types given in Signature on the incoming operand stack with the return type given in Signature, yielding the outgoing type state.

```
instructionIsTypeSafe(invokeinterface(CP, Count, 0), Environment, _Offset,
                StackFrame, NextStackFrame, ExceptionStackFrame) :-
        CP = imethod(MethodClassName, MethodName, Signature),
        MethodName \= '<init>',
        MethodName \= '<clinit>',
        parseMethodSignature(Signature, OperandArgList, ReturnType),
        reverse([class(MethodClassName) | OperandArgList], StackArgList),
```

```
    canPop(StackFrame, StackArgList, TempFrame),
    validTypeTransition(Environment, [], ReturnType, TempFrame, NextStackFrame),
    countIsValid(Count, StackFrame, TempFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The count operand of an `invokeinterface` instruction is valid if it is the difference between the size of the operand stack before and after the instruction executes.

```
countIsValid(Count, InputFrame, OutputFrame) :-
    InputFrame  = frame(_Locals1, OperandStack1, _Flags1),
    OutputFrame = frame(_Locals2, OperandStack2, _Flags2),
    length(OperandStack1, Length1),
    length(OperandStack2, Length2),
    Count =:= Length1 - Length2.
```

invokespecial:

An `invokespecial` instruction is type safe iff all of the following conditions hold:

- Its first operand, CP, refers to a constant pool entry denoting a method named MethodName with signature Signature that is a member of a class MethodClassName.

Either

- MethodName is not `<init>`.
- MethodName is not `<clinit>`.
- One can validly replace types matching the current class and the argument types given in Signature on the incoming operand stack with the return type given in Signature, yielding the outgoing type state.
- One can validly replace types matching the class MethodClassName and the argument types given in Signature on the incoming operand stack with the return type given in Signature.

```
instructionIsTypeSafe(invokespecial(CP), Environment, _Offset, StackFrame,
            NextStackFrame, ExceptionStackFrame) :-
    CP = method(MethodClassName, MethodName, Signature),
    MethodName \= '<init>',
    MethodName \= '<clinit>',
    parseMethodSignature(Signature, OperandArgList, ReturnType),
    thisClass(Environment, CurrentClass),
    reverse([CurrentClass | OperandArgList], StackArgList),
    validTypeTransition(Environment, StackArgList, ReturnType, StackFrame, NextStackFrame),
    reverse([class(MethodClassName) | OperandArgList], StackArgList2),
    validTypeTransition(Environment, StackArgList2, ReturnType, StackFrame,
                  _ResultStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

Or

- MethodName is `<init>`.
- Signature specifies a `void` return type.
- One can validly pop types matching the argument types given in Signature and an uninitialized type, UninitializedArg, off the incoming operand stack, yielding OperandStack.
- The outgoing type state is derived from the incoming type state by first replacing the incoming operand stack with OperandStack and then replacing all instances of UninitializedArg with the type of instance being initialized.

```
instructionIsTypeSafe(invokespecial(CP), Environment, _Offset, StackFrame,
            NextStackFrame, ExceptionStackFrame) :-
    CP = method(MethodClassName, '<init>', Signature),
    parseMethodSignature(Signature, OperandArgList, void),
    reverse(OperandArgList, StackArgList),
    canPop(StackFrame, StackArgList, TempFrame),
    TempFrame = frame(Locals, FullOperandStack, Flags),
    FullOperandStack = [UninitializedArg | OperandStack],
    rewrittenUninitializedType(UninitializedArg, Environment, class(MethodClassName), This),
    rewrittenInitializationFlags(UninitializedArg, Flags, NextFlags),
    substitute(UninitializedArg, This, OperandStack, NextOperandStack),
    substitute(UninitializedArg, This, Locals,       NextLocals),
    NextStackFrame      = frame(NextLocals, NextOperandStack, NextFlags),
    ExceptionStackFrame = frame(NextLocals, [], Flags).
```

Special rule for `invokespecial` of an `<init>` method.

This rule is the sole motivation for passing back a distinct exception stack frame. The concern is that `invokespecial` can cause a superclass `<init>` method to be invoked, and that invocation could fail, leaving `this` uninitialized. This situation cannot be created using Java programming language source code, but can be created through JVM assembly programming.

The original frame holds an uninitialized object in a local and has flag `uninitializedThis`. Normal termination of `invokespecial` initializes the uninitialized object and turns off the `uninitializedThis` flag. But if the invocation of an `<init>` method throws an exception, the uninitialized object might be left in a partially initialized state, and needs to be made permanently unusable. This is represented by an exception frame containing the broken object (the new value of the local) and the `uninitializedThis` flag (the old flag). There is no way to get from an apparently-initialized object bearing the `uninitializedThis` flag to a properly initialized object, so the object is permanently unusable. If not for this case, the exception stack frame could be the same as the input stack frame.

```
rewrittenUninitializedType(uninitializedThis, Environment, _MethodClass, This) :-
    thisClass(Environment, This).
```

```
rewrittenUninitializedType(uninitialized(Address), Environment, MethodClass, MethodClass) :-
    allInstructions(Environment, Instructions),
    member(instruction(Address, new(MethodClass)), Instructions).
```

Computes what type the uninitialized argument's type needs to be rewritten to.

There are 2 cases.

If we are initializing an object within its constructor, its type is initially `uninitializedThis`. This type will be rewritten to the type of the class of the `<init>` method.

The second case arises from initialization of an object created by `new`. The uninitialized arg type is rewritten to MethodClass, the type of the method holder of `<init>`. We check whether there really is a `new` instruction at Address.

```
rewrittenInitializationFlags(uninitializedThis, _Flags, []).
rewrittenInitializationFlags(uninitialized(_), Flags, Flags).


substitute(_Old, _New, [], []).
substitute(Old, New, [Old | FromRest], [New | ToRest]) :- substitute(Old, New, FromRest, ToRest).
substitute(Old, New, [From1 | FromRest], [From1 | ToRest]) :-
    From1 \= Old,
    substitute(Old, New, FromRest, ToRest).
```

invokestatic:

An `invokestatic` instruction is type safe iff all of the following conditions hold:

- Its first operand, CP, refers to a constant pool entry denoting a method named MethodName with signature Signature.
- MethodName is not `<clinit>`.
- One can validly replace types matching the argument types given in Signature on the incoming operand stack with the return type given in Signature, yielding the outgoing type state.

```
instructionIsTypeSafe(invokestatic(CP), Environment, _Offset, StackFrame,
            NextStackFrame, ExceptionStackFrame) :-
    CP = method(_MethodClassName, MethodName, Signature),
    MethodName \= '<clinit>',
    parseMethodSignature(Signature, OperandArgList, ReturnType),
    reverse(OperandArgList, StackArgList),
    validTypeTransition(Environment, StackArgList, ReturnType, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

invokevirtual:

An `invokevirtual` instruction is type safe iff all of the following conditions hold:

- Its first operand, CP, refers to a constant pool entry denoting a method named MethodName with signature Signature that is a member of an class MethodClassName.
- MethodName is not `<init>`.
- MethodName is not `<clinit>`.
- One can validly replace types matching the class MethodClassName and the argument types given in Signature on the incoming operand stack with the return type given in Signature, yielding the outgoing type state.

```
instructionIsTypeSafe(invokevirtual(CP), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        CP = method(MethodClassName, MethodName, Signature),
        MethodName \= '<init>',
        MethodName \= '<clinit>',
        parseMethodSignature(Signature, OperandArgList, ReturnType),
        reverse([class(MethodClassName) | OperandArgList], StackArgList),
        validTypeTransition(Environment, StackArgList, ReturnType, StackFrame,
                        NextStackFrame),
        passesProtectedCheck(Environment, MethodClassName, MethodName, Signature,
                            StackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ior:

An `ior` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(ior, iadd).
```

irem:

An `irem` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(irem, iadd).
```

ireturn:

An `ireturn` instruction is type safe if the enclosing method has a declared return type of `int`, and one can validly pop a type matching `int` off the incoming operand stack.

```
instructionIsTypeSafe(ireturn, Environment, _Offset, StackFrame,
              afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, int),
    canPop(StackFrame, [int], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ishl:

An `ishl` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(ishl, iadd).
```

ishr:

An `ishr` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(ishr, iadd).
```

istore:

An `istore` instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a store instruction with operand Index and type `int` is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(istore(Index), Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

istore_<n>:

The instructions `istore_<n>`, for $0 \leq n \leq 3$, are typesafe iff the equivalent `istore` instruction is type safe.

```
instructionHasEquivalentTypeRule(istore_0, istore(0)).
instructionHasEquivalentTypeRule(istore_1, istore(1)).
instructionHasEquivalentTypeRule(istore_2, istore(2)).
instructionHasEquivalentTypeRule(istore_3, istore(3)).
```

isub:

An `isub` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(isub, iadd).
```

iushr:

An `iushr` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(iushr,iadd).
```

ixor:

An `ixor` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(ixor, iadd).
```

l2d:

An `l2d` instruction is type safe if one can validly pop `long` off the incoming operand stack and replace it with `double`, yielding the outgoing type state.

```
instructionIsTypeSafe(l2d, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [long], double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

l2f:

An `l2f` instruction is type safe if one can validly pop `long` off the incoming operand stack and replace it with `float`, yielding the outgoing type state.

```
instructionIsTypeSafe(l2f, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [long], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

l2i:

An `l2i` instruction is type safe if one can validly pop `long` off the incoming operand stack and replace it with `int`, yielding the outgoing type state.

```
instructionIsTypeSafe(l2i, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [long], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ladd:

An `ladd` instruction is type safe iff one can validly replace types matching `long` and `long` on the incoming operand stack with `long` yielding the outgoing type state.

```
instructionIsTypeSafe(ladd, Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [long, long], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

laload:

An `laload` instruction is type safe iff one can validly replace types matching `int` and array of `long` on the incoming operand stack with `long` yielding the outgoing type state.

```
instructionIsTypeSafe(laload, Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int, arrayOf(long)], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

land:

An `land` instruction is type safe iff the equivalent `ladd` instruction is type safe.

```
instructionHasEquivalentTypeRule(land, ladd).
```

lastore:

A `lastore` instruction is type safe iff one can validly pop types matching `long, int` and array of `long` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(lastore, _Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [long, int, arrayOf(long)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lcmp:

A `lcmp` instruction is type safe iff one can validly replace types matching `long` and `long` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(lcmp, Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [long, long], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lconst_<l>:

An `lconst_0` instruction is type safe if one can validly push the type `long` onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(lconst_0, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

An `lconst_1` instruction is type safe iff the equivalent `lconst_0` instruction is type safe.

```
instructionHasEquivalentTypeRule(lconst_1,lconst_0).
```

ldc:

An `ldc` instruction with operand CP is type safe iff CP refers to a constant pool entry denoting an entity of type Type, where Type is either `int`, `float` or `String`, and one can validly push Type onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(ldc(CP), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    functor(CP, Tag, _),
    member([Tag, Type], [[int,   int], [float, float], [string, class('java/lang/String')]]),
    validTypeTransition(Environment, [], Type, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ldc_w:

An `ldc_w` instruction is type safe iff the equivalent `ldc` instruction is type safe.

```
instructionHasEquivalentTypeRule(ldc_w(CP), ldc(CP))
```

ldc2_w:

An `ldc2_w` instruction with operand CP is type safe iff CP refers to a constant pool entry denoting an entity of type Tag, where Tag is either `long` or `double`, and one can validly push Tag onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(ldc2_w(CP), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    functor(CP, Tag, _),
    member(Tag, [long, double]),
    validTypeTransition(Environment, [], Tag, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ldiv:

An `ldiv` instruction is type safe iff the equivalent `ladd` instruction is type safe.

```
instructionHasEquivalentTypeRule(ldiv, ladd).
```

lload:

An `lload` instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a load instruction with operand Index and type `long` is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(lload(Index), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lload_<n>:

The instructions `lload_<n>`, for $0 \le n \le 3$, are typesafe iff the equivalent `lload` instruction is type safe.

```
instructionHasEquivalentTypeRule(lload_0, lload(0)).
instructionHasEquivalentTypeRule(lload_1, lload(1)).
instructionHasEquivalentTypeRule(lload_2, lload(2)).
instructionHasEquivalentTypeRule(lload_3, lload(3)).
```

lmul:

An `lmul` instruction is type safe iff the equivalent `ladd` instruction is type safe.

```
instructionHasEquivalentTypeRule(lmul, ladd).
```

lneg:

An `lneg` instruction is type safe iff there is a type matching `long` on the incoming operand stack. The `lneg` instruction does not alter the type state.

```
instructionIsTypeSafe(lneg, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [long], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lookupswitch:

A `lookupswitch` instruction is type safe if its keys are sorted, one can validly pop `int` off the incoming operand stack yielding a new type state BranchStackFrame, and all of the instructions targets are valid branch targets assuming BranchStackFrame as their incoming type state.

```
instructionIsTypeSafe(lookupswitch(Targets, Keys), Environment, _, StackFrame,
                afterGoto, ExceptionStackFrame) :-
    sort(Keys, Keys),
    canPop(StackFrame, [int], BranchStackFrame),
    checklist(targetIsTypeSafe(Environment, BranchStackFrame), Targets),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lor:

An `lor` instruction is type safe iff the equivalent `ladd` instruction is type safe.

```
instructionHasEquivalentTypeRule(lor, ladd).
```

lrem:

An `lrem` instruction is type safe iff the equivalent `ladd` instruction is type safe.

```
instructionHasEquivalentTypeRule(lrem, ladd).
```

lreturn:

An `lreturn` instruction is type safe if the enclosing method has a declared return type of `long`, and one can validly pop a type matching `long` off the incoming operand stack.

```
instructionIsTypeSafe(lreturn, Environment, _Offset, StackFrame,
                afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, long),
    canPop(StackFrame, [long], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lshl:

An `lshl` instruction is type safe if one can validly replace the types `int` and `long` on the incoming operand stack with the type `long` yielding the outgoing type state.

```
instructionIsTypeSafe(lshl, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int, long], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lshr:

An lshr instruction is type safe iff the equivalent lshl instruction is type safe.

```
instructionHasEquivalentTypeRule(lshr, lshl).
```

lstore:

An lstore instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a store instruction with operand Index and type long is type safe and yields an outgoing type state NextStackFrame.

```
instructionIsTypeSafe(lstore(Index), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lstore_<n>:

The instructions lstore_<n>, for $0 \le n \le 3$, are typesafe iff the equivalent lstore instruction is type safe.

```
instructionHasEquivalentTypeRule(lstore_0, lstore(0)).
instructionHasEquivalentTypeRule(lstore_1, lstore(1)).
instructionHasEquivalentTypeRule(lstore_2, lstore(2)).
instructionHasEquivalentTypeRule(lstore_3, lstore(3)).
```

lsub:

An lsub instruction is type safe iff the equivalent ladd instruction is type safe.

```
instructionHasEquivalentTypeRule(lsub,    ladd).
```

lxor:

An lxor instruction is type safe iff the equivalent ladd instruction is type safe.

```
instructionHasEquivalentTypeRule(lxor,    ladd).
```

lushr:

An lushr instruction is type safe iff the equivalent lshl instruction is type safe.

```
instructionHasEquivalentTypeRule(lushr,    lshl).
```

monitorenter:

A `monitorenter` instruction is type safe iff one can validly pop a type matching `reference` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(monitorenter, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [reference], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

monitorexit:

A `monitorexit` instruction is type safe iff the equivalent `monitorenter` instruction is type safe.

```
instructionHasEquivalentTypeRule(monitorexit, monitorenter).
```

multinewarray:

A `multinewarray` instruction with operands CP and Dim is type safe iff CP refers to a constant pool entry denoting an array type whose dimension is greater or equal to Dim, Dim is strictly positive, and one can validly replace Dim `int` types on the incoming operand stack with the type denoted by CP yielding the outgoing type state.

```
instructionIsTypeSafe(multianewarray(CP, Dim), Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
        CP = arrayOf(_),
        classDimension(CP, Dimension),
        Dimension >= Dim,
        Dim > 0,
        /* Make a list of Dim ints */
        findall(int, between(1, Dim, _), IntList),
        validTypeTransition(Environment, IntList, CP, StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The dimension of an array type whose component type is also an array type is 1 more than the dimension of its component type.

```
classDimension(arrayOf(X), Dimension) :-
        classDimension(X, Dimension1),
        Dimension is Dimension1 + 1.
        classDimension(_, Dimension) :- Dimension = 0.
```

new:

A `new` instruction with operand CP at offset Offset is type safe iff CP refers to a constant pool entry denoting a class type, the type `uninitialized(Offset)` does not appear in the incoming operand stack, and one can validly push `uninitialized(Offset)` onto the incoming operand stack and replace `uninitialized(Offset)` with `top` in the incoming local variables yielding the outgoing type state.

```
instructionIsTypeSafe(new(CP), Environment, Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
      StackFrame = frame(Locals, OperandStack, Flags),
      CP = class(_),
      NewItem = uninitialized(Offset),
      notMember(NewItem, OperandStack),
      substitute(NewItem, top, Locals, NewLocals),
      validTypeTransition(Environment, [], NewItem,
                          frame(NewLocals, OperandStack, Flags), NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

newarray:

A `newarray` instruction with operand TypeCode is type safe iff TypeCode corresponds to the primitive type ElementType, and one can validly replace the type `int` on the incoming operand stack with the type array of ElementType, yielding the outgoing type state.

```
instructionIsTypeSafe(newarray(TypeCode), Environment, _Offset, StackFrame,
              NextStackFrame, ExceptionStackFrame) :-
      primitiveArrayInfo(TypeCode, _TypeChar, ElementType, _VerifierType),
      validTypeTransition(Environment, [int], arrayOf(ElementType), StackFrame,
                          NextStackFrame),
      exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The correspondence between type codes and primitive types is specified by the following predicate:

```
primitiveArrayInfo(4,  0'Z, boolean,  int).
primitiveArrayInfo(5,  0'C, char,     int).
primitiveArrayInfo(6,  0'F, float,    float).
primitiveArrayInfo(7,  0'D, double,   double).
primitiveArrayInfo(8,  0'B, byte,     int).
primitiveArrayInfo(9,  0'S, short,    int).
primitiveArrayInfo(10, 0'I, int,      int).
primitiveArrayInfo(11, 0'J, long,     long).
```

nop:

A nop instruction is always type safe. The nop instruction does not affect the type state.

```
instructionIsTypeSafe(nop, _Environment, _Offset, StackFrame,
                      StackFrame, ExceptionStackFrame) :-
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

pop:

A pop instruction is type safe iff one can validly pop a category 1 type off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(pop, _Environment, _Offset, StackFrame,
                        NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, [Type | Rest], Flags),
    Type \= top,
    sizeOf(Type, 1),
    NextStackFrame = frame(Locals, Rest, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

pop2:

A pop2 instruction is type safe iff it is a type safe form of the pop2 instruction.

```
instructionIsTypeSafe(pop2, _Environment, _Offset, StackFrame,
                      NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    pop2SomeFormIsTypeSafe(InputOperandStack, OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A pop2 instruction is *a type safe form of the* pop2 *instruction* iff it is a type safe form 1 pop2 instruction or a type safe form 2 pop2 instruction.

```
pop2SomeFormIsTypeSafe(InputOperandStack, OutputOperandStack) :-
    pop2Form1IsTypeSafe(InputOperandStack, OutputOperandStack).

pop2SomeFormIsTypeSafe(InputOperandStack, OutputOperandStack) :-
    pop2Form2IsTypeSafe(InputOperandStack, OutputOperandStack).
```

A pop2 instruction is *a type safe form 1* pop2 *instruction* iff one can validly pop two types of size 1 off the incoming operand stack yielding the outgoing type state.

```
pop2Form1IsTypeSafe([Type1, Type2 | Rest], Rest) :-
    sizeOf(Type1, 1),
    sizeOf(Type2, 1).
```

A pop2 instruction is *a type safe form 2* pop2 *instruction* iff one can validly pop a type of size 2 off the incoming operand stack yielding the outgoing type state.

```
pop2Form2IsTypeSafe([top, Type | Rest], Rest) :-
    sizeOf(Type, 2).
```

putfield:

A putfield instruction with operand CP is type safe iff CP refers to a constant pool entry denoting a field whose declared type is FieldType, declared in a class FieldClass, and one can validly pop types matching FieldType and FieldClass off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(putfield(CP), _Environment, _Offset, StackFrame,
            NextStackFrame, ExceptionStackFrame) :-
    CP = field(FieldClass, FieldName, FieldSignature),
    parseFieldSignature(FieldSignature, FieldType),
    passesProtectedCheck(Environment, FieldClass, FieldName, FieldSignature,
                        StackFrame),
    canPop(StackFrame, [FieldType, class(FieldClass)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

putstatic:

A putstatic instruction with operand CP is type safe iff CP refers to a constant pool entry denoting a field whose declared type is FieldType, and one can validly pop a type matching FieldType off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(putstatic(CP), _Environment, _Offset, StackFrame,
            NextStackFrame, ExceptionStackFrame) :-
    CP = field(_FieldClass, _FieldName, FieldSignature),
    parseFieldSignature(FieldSignature, FieldType),
    canPop(StackFrame, [FieldType], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

return:

A `return` instruction is type safe if the enclosing method declares a `void` return type, and either:

- The enclosing method is not an `<init>` method, **or**
- `this` has already been completely initialized at the point where the instruction occurs.

```
instructionIsTypeSafe(return, Environment, _Offset, StackFrame,
                afterGoto, ExceptionStackFrame) :-
        thisMethodReturnType(Environment, void),
        StackFrame = frame(_Locals, _OperandStack, Flags),
        notMember(flagThisUninit, Flags),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

saload:

An `saload` instruction is type safe iff one can validly replace types matching `int` and array of `short` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(saload, Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        validTypeTransition(Environment, [int, arrayOf(short)], int, StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

sastore:

An `sastore` instruction is type safe iff one can validly pop types matching `int, int` and array of `short` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(sastore, _Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        canPop(StackFrame, [int, int, arrayOf(short)], NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

sipush:

An `sipush` instruction is type safe iff one can validly push the type `int` onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(sipush(_Value), Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
        validTypeTransition(Environment, [], int, StackFrame, NextStackFrame),
        exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

swap:

A `swap` instruction is type safe iff one can validly replace two category 1 types, Type1 and Type2, on the incoming operand stack with the types Type2 and Type1 yielding the outgoing type state.

```
instructionIsTypeSafe(swap, _Environment, _Offset, StackFrame,
                NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(_Locals, [Type1, Type2 | Rest], _Flags),
    sizeOf(Type1, 1),
    sizeOf(Type2, 1),
    NextStackFrame = frame(_Locals, [Type2, Type1 | Rest], _Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

tableswitch:

A `tableswitch` instruction is type safe if its keys are sorted, one can validly pop `int` off the incoming operand stack yielding a new type state BranchStackFrame, and all of the instructions targets are valid branch targets assuming BranchStackFrame as their incoming type state.

```
instructionIsTypeSafe(tableswitch(Targets, Keys), Environment, _Offset,
                StackFrame, afterGoto, ExceptionStackFrame) :-
    sort(Keys, Keys),
    canPop(StackFrame, [int], BranchStackFrame),
    checklist(targetIsTypeSafe(Environment, BranchStackFrame), Targets),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

wide:

The `wide` instructions follow the same rules as the instructions they widen.

```
instructionHasEquivalentTypeRule(wide(WidenedInstruction), WidenedInstruction).
```

The type state after an instruction completes abruptly is the same as the incoming type state, except that the operand stack is empty.

```
exceptionStackFrame(StackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, _OperandStack, Flags),
    ExceptionStackFrame = frame(Locals, [], Flags).
```

Most of the type rules in this specification depend on the notion of a valid type transition

A type transition is valid if one can pop a list of expected types off the incoming type state's operand stack and replace them with an expected result type, resulting in a new valid type state. In particular, the size of the operand stack in the new type state must not exceed its maximum declared size.

```
validTypeTransition(Environment, ExpectedTypesOnStack, ResultType,
            frame(Locals, InputOperandStack, Flags),
            frame(Locals, NextOperandStack, Flags)) :-
    popMatchingList(InputOperandStack, ExpectedTypesOnStack, InterimOperandStack),
    pushOperandStack(InterimOperandStack, ResultType, NextOperandStack),
    operandStackHasLegalLength(Environment, NextOperandStack).
```

Access Ith element of the operand stack from a type state.

```
nth1OperandStackIs(I, frame(_Locals, OperandStack, _Flags), Element) :-
    nth1(I, OperandStack, Element).
```

4. References

William F. Clocksin and Christopher S. Mellish. *Programming in Prolog, Fourth Edition.* Springer-Verlag, 1994.

Leon Sterling and Ehud Shapiro. *The Art of Prolog, Second Edition.* MIT Press, 1994.

Timothy Lindholm and Frank Yellin. *The Java[TM] Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.

Sheng Liang. *The KVM Verifier*. Unpublished internal Sun document.

Frank Yellin. "Low Level Security in Java." *World Wide Web Journal*, Volume I, Issue 1, Winter 1996. Available online at `http://www.w3journal.com/1/f.197/paper/197.html`.