Dipartimento di Informatica e
Scienze dell'Informazione

Type systems for Java separate compilation and selective recompilation

by

Giovanni Lagorio

Theses Series **DISI-TH-2004-XX**

DISI, Università di Genova

v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

Dipartimento di Informatica e
Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

Type systems for Java separate compilation and selective recompilation

by

Giovanni Lagorio

March, 2004

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Univ. di Genova
via Dodecaneso 35
I-16146 Genova, Italy
`http://www.disi.unige.it/`

Ph.D. Thesis in Computer Science

Submitted by Giovanni Lagorio
DISI, Univ. di Genova
`lagorio@disi.unige.it`

Date of submission: March 2004

Title: Type systems for Java separate compilation and
selective recompilation

Advisors:
Davide Ancona
DISI, Univ. di Genova
`davide@disi.unige.it`

Elena Zucca
DISI, Univ. di Genova
`zucca@disi.unige.it`

Supervisor: Elena Zucca

Ext. Reviewers:

Gilad Bracha
Sun Microsystems
`gilad@bracha.org`

Sophia Drossopoulou
Imperial College of Science, Technology and Medicine, University of London
`sd@doc.ic.ac.uk`

Paola Giannini
Università del Piemonte Orientale
`giannini@di.unito.it`

Abstract

In this thesis we provide a formal framework for separate compilation of Java-like languages. We start by formalizing the Java compilation process as it is currently implemented by compilers. Indeed, the behavior of standard compilers is hard to be understood, since compilation is propagated in a complex way to sources not explicitly mentioned by the user, and a specification is completely missing. Furthermore, with this formalization we show that Java separate compilation, despite its complexity, does not satisfy some desirable properties, such as *safety* and *equivalence with global compilation*. The former states that binaries produced by a successful compilation should not throw linkage exceptions when run; the latter states that the recompilation of a subset of the existing fragments should be equivalent to the recompilation of the whole program.

Therefore, we show how to get these properties. First, we achieve safety by modifying the Java compilation process, still keeping the approach of current compilers and existing type systems, where a single Java fragment is typechecked w.r.t. a type environment containing full type information on used fragments, which is extracted from the compilation context.

Then, we introduce an innovative type system where a fragment is typechecked in total isolation, w.r.t. a type environment consisting in fine-grained assumptions which describe the minimal requirements needed for generating a given binary. This system can be successfully exploited to define a recompilation strategy which is both *sound*, that is, equivalent to global recompilation, and *optimal*, that is, s.t. a source is *never* recompiled if not necessary.

In order to provide a solid starting point for implementing a compilation manager based on these ideas, we extend the type system with most of the Java-specific features, showing why they hinder a sound and optimal selective recompilation mechanism and how they can be successfully handled. Finally, we discuss implementation issues.

Voodoo Programming: Things programmers do that they know shouldn't work but they try anyway, and which sometimes actually work, such as recompiling everything. (Karl Lehenbauer)

Acknowledgements

First of all, I'd like to express my gratitude to my parents, who have always trusted and supported me.

My advisors Elena and Davide deserve special thanks for their endless patience and support, from both the technical and the "philosophical" points of view. Their love for research in seeking the Truth is infectious, and I'm glad I've been "infected" :-)

During my PhD I've visited Imperial College, which has been a wonderful experience. I'm very grateful to Sophia for making it possible, for her kindness and help. I'd also like to thank Susan and all the other skilled guys who attended SLURP meetings during my stay in London: every brainstorming session was fun and fruitful. The contents of this thesis has been surely enhanced by some of the ideas discussed there.

For all the time we have shared, their help and encouragement I'd like to thank Walter, Lalla, Sonia and all the other inhabitants of planet DISI ;)

Last, but not least, a very warm thank-you to Fabio and Chiara, for being the friends I can always count on.

Contents

List of Figures

Chapter 1

Introduction

In modern programming languages, the notion of "program" as a whole has become more and more obsolete. Nowadays, the process of developing software typically consists in writing separate pieces of code, which we call *fragments*, following [Car97], each one implementing some basic functionality and relying on functionalities provided by other fragments.

A language should provide facilities which allow the development of fragments to be as much modular as possible. In particular, a highly desirable feature is *separate compilation*, which means, in its strongest formulation, the possibility of typechecking and compiling a single source fragment `S` in isolation, that is, in a context where only type information but no code is available on the fragments `S` depends on.

Two properties are desirable to exploit the power of separate compilation without risking counterintuitive semantics:

- type safety;
- equivalence with global (re)compilation.

The former guarantees that the execution of a program, which is the result of a successful compilation, never throws linkage related exceptions (as `NoSuchMethodError`).

The latter expresses the intuitive idea of being able to compile different fragments at different times, obtaining, in an incremental way, the same overall result that would be obtained recompiling all the fragment at once (being that result either a set of binaries or an error).

Even though Java is known to support separate compilation, neither type safety nor equivalence to global compilation are met.

1.1 Type safety

As the following example shows, in Java it may happen that the execution of an application, which is the result of a successful compilation, throws a linkage error.

Assume to have the following declarations in two source fragments named `A.java` and `B.java`.

```
class A {
    void f(B b) {
        if (b!=null)
            b.g(this) ;
    }
    void h() {}
    public static void main(String [] args) {
        new A().f ( args.length > 0 ? new B() : null ) ;
    }
}
class B {
    void g(A a) {
         a.h() ;
    }
}
```

We can compile these fragments obtaining the corresponding binaries `A.class` and `B.class`.

Suppose now to remove the method `h` from class `A`, and then to invoke the compilation on `A` in the current context (where `B.class` is available). The compilation succeeds, because the binary `B.class` is newer than its source `B.java`, so the compilation is not automatically propagated to `B.java`, and the type information needed to typecheck the declaration of class `A` are extracted from the binary `B.class`.[1]

At run-time the lack of method `h` can either pass unnoticed (if the method is never invoked) or cause a run-time error, modeled in Java by the exception `NoSuchMethodError`, if there is an attempt at calling the method.

In this particular case, if we run class `A` without arguments then no exception is thrown and the program terminates (successfully); on the other hand, passing any argument to the program causes the exception `NoSuchMethodError` to be thrown. Therefore, we can conclude that standard Java separate compilation is not type safe. This issue, well-known

[1]Note that, on the other hand, the compilation of both sources would fail because `B` tries to invoke the method `h()`, which is no longer available in `A`.

to experienced Java programmers, seems in contradiction with the fact that type safety results have been proved for the Java language [vON99, DE99, Sym99]; the explanation is that these formal type systems, and the related type safety results, are only related to the special case when a closed set of source fragments is typechecked. However, always taking this approach would make totally useless having a support for separate compilation.

1.2 Equivalence with global recompilation

Because sources depend on each other, a change to a source may affect the result of the compilation of other *unchanged* sources. For this reason, the compilation of a set of (changed) sources may require the recompilation of other (unchanged) sources to obtain the same result a global recompilation would produce.

Even though Java compilers in *some* cases propagate the compilation to fragments not explicitly mentioned when invoking the compiler, they fail to guarantee equivalence with a global recompilation. In particular, in Java it may happen that:

1. the recompilation of a set of changed sources succeeds, while the recompilation of the whole program fails;
2. the recompilation of a set of changed sources fails, while the recompilation of the whole program succeeds;
3. both recompilations succeed, but they produce different binaries, hence they lead to different executions.

We have already shown an example of the first problem in Section 1.1; indeed, in that example the recompilation of the single class `A` succeeds, while a global recompilation would fail. So, let us consider the other two cases.

The following example shows that the recompilation of a subset of the sources may fail while the compilation of the whole program succeeds. Assume to have the following declarations in two source fragments named `A.java` and `B.java`.

```
class A {
    final static boolean b = B.b ;
}
class B {
    final static boolean b = false ;
}
```

We can compile these fragments obtaining the corresponding bytecode fragments `A.class` and `B.class`. Suppose now to modify `B.java` in the following way:

```
class B {
    final static boolean b = true ;
    void f() {
        while (A.b) {
            System.out.println("Hello, world!") ;
        }
    }
}
```

The compilation of the single class `B` would fail, because the constant `A.b` found in `A.class` is `false` and thus the body of the `while` statement turns out to be unreachable[2]; note that in this case, differently from the preceding, `B` depends on `A`, hence, the compilation of `B` would require the compilation of `A` in the absence of the bytecode for `A`; however, even in a context where the bytecode for `A` is available (as in this case), the compiler would not recompile `A.java` because the class file turns out to be newer than the corresponding source.

On the other hand, the compilation of both source fragments would succeed because in this case the value of `A.b` would be `true`.

As the reader has probably noticed, this counterexample relies on two quite peculiar features of Java:

- a standard Java compiler must raise an error when some part of code turns out to be unreachable[3];
- a `final` field initialized with a compile-time constant is itself a compile-time constant[4], hence, during compilation, it is bound to the calculated value, thus losing the dependency information.

We analyze both issues in detail in Chapter 4.

The following example shows a case where both the recompilation of a subset of the sources and the recompilation of the whole program succeed, but producing different bytecode fragments (hence, different run-time semantics). Assume to have the following declarations in two source fragments named `A.java` and `B.java`.

[2] See 14.20 and 15.28 of [GJSB00].

[3] Except for `if` statements that are treated in a special way to permit conditional compilation, see 14.20 of [GJSB00]

[4] See 15.28 of [GJSB00].

```
class A {
    void f(B b) {
        System.out.println( "The answer is: "+b.g(b) ) ;
    }
    public static void main(String [] argv) {
        new A().f(new B()) ;
    }
}
class B extends A {
    int g(A a) { return 1 ; }
}
```

We can compile these fragments obtaining the corresponding bytecode fragments `A.class` and `B.class`. When we run `A.class` we obtain:

```
The answer is: 1
```

Assume now to add an overloaded method to class `B`, obtaining the following `B.java`:

```
class B extends A {
    int g(A a) { return 1 ; }
    int g(B b) { return 42 ; }
}
```

If we recompile (successfully) class `B` and then re-run `A.class`, then we obtain the same output as before, but if we recompile both source fragments and run `A.class` we obtain the different output shown below.

```
The answer is: 42
```

The example above is based on the fact that overloading resolution is performed at compile-time. Hence, while in a source fragment the method which is selected for a given invocation depends on the context, that is, on the methods which are currently provided by the class of the receiver, at the binary level this dependency is lost (as in the case of `final` fields initialized with compile-time constants mentioned above) and the invoked method is fixed once and for all. The same happens in all the situations where the resolution is static, like, e.g., field accesses.

1.3 Java-like separate compilation

Although there is a specification which describes how to compile Java language into its bytecode representation (to be executed by the Java Virtual Machine), there is no specification of Java separate compilation. So, the semantics of separate compilation is implementation dependent and, as we have seen, in some cases it is not type-safe (nor equivalent to a global recompilation).

In order to study Java-like separate compilation and its properties we introduce, in Chapter 2, a formal framework providing a rigorous basis for:

- defining and investigating different possibilities for the overall compilation process as in Java (for instance: a minimal set of checks, the checks performed by standard Java compilers, as many checks as possible)
- proving desirable properties, like type safety, for a compilation process.

The framework is based on the notion of *compilation schema*. A compilation schema consists of four components:

- a source type judgment, which models the compilation (that is, typechecking and code generation) of a source fragment;
- a binary type judgment, which models the typechecking of a binary fragment;
- a dependency function, which models the fact that the compilation of a fragment may enforce the compilation of other fragments;
- a type extraction function, which extracts from a compilation environment (a collection of source and binary fragments) a type environment providing the type information needed for typechecking fragments.

The first two components model the part of Java compilation which corresponds to truly separate compilation in the sense of [Car97], although, as the presence of a type extraction function hints, in the Java-like approach a fragment cannot be compiled in total isolation. Indeed, fragments are always checked against a type environment extracted from other fragments, which must be present (at least in binary form).

The language we consider is reminiscent of Featherweight Java [IPW99], in the sense that it is a small functional subset of Java; however, since here we are also interested in code generation and bytecode execution, we present a simple binary language as well, together with its reduction semantics. Our description of bytecode is rather abstract: we basically enrich the source language with some annotations. For instance, each method invocation is annotated with a method descriptor which describes the method which has been statically selected for the invocation.

We consider three different compilation schemata. While all these schemata share the same source type judgment and type extraction function (corresponding to the Java type system defined in [GJSB00]), they remarkably differ in the other two components (that is, dependency function and binary type judgment).

The first schema we consider, which we call *minimal*, corresponds to true separate type-checking, in the sense that typechecking is not propagated. However, as noted, compiling a source `S` requires the availability of fragments `S` depends on, since some of the type information needed for typechecking `S` have to be extracted from them.

The second compilation schema, that we call SDK, is the one used by standard Java compilers (at least for what we have been able to understand by experiments, since no specification is available). In this case invoking compilation on `S` enforces typechecking of other fragments, but not of all those which could be possibly loaded at run-time and, moreover, no checks are performed on binary fragments. As a consequence, binaries obtained as result of the compilation are not guaranteed to link safely at run-time.

Finally, we propose a compilation schema which is type safe, that is, guarantees safe linking at run-time. For this last schema we provide a full definition of the four components and we prove type safety. That is, we prove that executions started from classes that are the product of the compilation do not throw linkage errors.

This last result shows that type safety can be achieved by keeping a Java-like separate compilation (that is, an approach where a class is not compiled in total isolation, but checked against a type environment extracted from the current compilation context), by a modification of two components of the SDK compilation schema. That is, having stronger dependencies which propagate typechecking to all classes which could be possibly loaded at run-time, and a non trivial binary type judgment.

The other property we are interested in, that is, equivalence with global recompilation, could also be achieved by having stronger dependencies. Roughly, when recompiling a class after a change, we could propagate typechecking to all classes which use either directly or indirectly this class. Notwithstanding this recompilation strategy would be equivalent to global recompilation, we could perform useless recompilations, since we would recompile a class `C'` which uses a class `C` *whichever is* the change made to `C`, regardless this change affects typechecking of `C'` or not. In practice, this strategy would be no much better than just recompiling all classes each time a change is made.

In order to obtain a better recompilation strategy, we need to take an approach different from Java-like separate compilation, which is illustrated in the following section.

1.4 True separate compilation and selective recompilation

In Chapter 3 we strive for a compilation which is truly separate, introducing an innovative type system where a fragment is typechecked in total isolation. As the type environment the fragment is compiled against is not extracted from the other fragments, in this system the compilation of a source closely corresponds to what Cardelli [Car97] calls *intra-checking* phase.

In this type system we introduce type environments which, differently from *standard* type environments used in Java-like compilations, are collections of type constraints describing fine grain requirements. For instance, let us consider the following class declaration:

```
class H extends P {
  int g(P p) {
    return p.f(new H()) ;
  }
  int m() {
    return new H().g(new P()) ;
  }
  U id(U u) {
    return u ;
  }
  X em(Y y) {
    return y ;
  }
}
```

and analyze under which assumptions class `H` can be successfully compiled. If we take the approach of Java-like separate compilation previously described, then we would need to impose rather strong requirements on all classes used by `H`, by asking for the most detailed type information about such classes.

Formally, this corresponds to compile `H` in a type environment Γ^{S} ("S" stands for standard) which contains assumptions on the types of classes `P`, `U`, `X` and `Y`. For instance, we can compile `H` in the type environment

$$\Gamma^{\mathrm{S}} = \mathtt{P} \mapsto <\mathtt{Object}, \mathtt{int\ f(Object)}>, \mathtt{U} \mapsto <\mathtt{Object}, >, \mathtt{Y} \mapsto <\mathtt{X}, >, \mathtt{X} \mapsto <\mathtt{Object}, >$$

corresponding to assume that class `P` extends `Object` and declares only `int f(Object)`, classes `U` and `X` both extend `Object` and are empty, and class `Y` extends `X` and is empty.

Under the assumptions contained in Γ^{S} class `H` can be successfully compiled into the following binary fragment $\mathtt{B_h}$:

```
class H extends P {
  int g(P p) {
    return p.f<<P.int (Object)>>(new H()) ;
  }
  int m() {
    return new H().g<<H.int (P)>>(new P()) ;
  }
  U id(U u) {
    return u ;
  }
  X em(Y y) {
    return y ;
  }
}
```

In our language a binary fragment is just like a source fragment except that method invocations contain an *annotation* $\ll \mathtt{C.T}\ (\mathtt{T_1} \ldots \mathtt{T_n}) \gg$ giving the class of the receiver C (in which the method is to be found at run-time, see 5.1 of [LY99]), the return type T, and the parameter types $\mathtt{T}_1 \ldots \mathtt{T}_n$ of the method which has been selected as most specific at compile time.

Let us now try to relax the strong assumptions in Γ^{S} by seeking an environment Γ^{NS} ("NS" stands for non-standard) containing other kinds of type assumptions which still guarantee that H compiles to the same binary fragment $\mathtt{B_h}$, but impose fairly weaker requirements on classes P, U, X and Y.

A first basic request is that the compilation environment containing H must provide a definition for the four classes which H depends on. In our system the existence of class C is expressed by a nonstandard assumption of the form $\exists\,\mathtt{C}$, therefore Γ^{NS} will contain at least the assumptions $\exists\,\mathtt{P}, \exists\,\mathtt{U}, \exists\,\mathtt{X}, \exists\,\mathtt{Y}$.

Let us now focus on the other assumptions needed for compiling class H into $\mathtt{B_h}$.

Class P: the following additional assumptions on class P are needed:

- $\mathtt{P} \not\leq \mathtt{H}$: P cannot be a subtype of H (since inheritance cannot be cyclic).
- P☺int g(P): P can be correctly extended with method `int g(P)`; that is, according to Java rules on method overriding, if P has a method g(P), then g must have the same return type int as declared in H. Analogous requirements are needed for the other methods declared in H.
- $\mathtt{P.f(H)} \overset{\mathtt{res}}{\rightarrow}$ <Object, int>: invocation of method f on an object of type P with an argument of type H is successfully resolved to a method with a parameter of type

Object and return type int. This assumption ensures that the body of g in H is successfully compiled to the same bytecode as in $\mathtt{B_h}$ (in other words, the same symbolic reference to the method is generated).

Class U: no additional requirements on U are needed, since the static correctness of method id in H only requires the existence of U.
Classes X and Y: class Y must be a subtype of class X, otherwise method em in H would not be statically correct. Therefore we need to add the assumption $\mathtt{Y} \leq \mathtt{X}$.

In conclusion, class H can be successfully compiled to $\mathtt{B_h}$ in the environment Γ^{NS} defined by:

$$\Gamma^{\text{NS}} = \exists\,\mathtt{P}, \exists\,\mathtt{U}, \exists\,\mathtt{X}, \exists\,\mathtt{Y}, \mathtt{P} \not\leq \mathtt{H}, \mathtt{P}☺\mathtt{int\ g(P)}, \mathtt{P}☺\mathtt{int\ m()},$$
$$\mathtt{P}☺\mathtt{U\ id(U)}, \mathtt{P}☺\mathtt{X\ em(Y)}, \mathtt{P.f(H)} \stackrel{\mathtt{res}}{\to} <\mathtt{Object}, \mathtt{int}>, \mathtt{Y} \leq \mathtt{X}$$

Note that Γ^{NS} is *weaker* than Γ^{S}, in the sense that it imposes less restrictive constraints; for instance, class U can extend any class and declare any method in Γ^{NS}, while class U must extend Object and be empty in Γ^{S}.

While the ability of compiling a source S in total isolation is appealing, the direct use of this type system would put the burden of writing the type assumptions on programmers. This activity is both tedious and rather time-consuming for real projects. Fortunately, the type assumptions for compiling a particular source S (to a particular binary B) can be automatically extracted, if the compilation of S is performed against a standard type environment.

The key point is that these automatically generated assumptions Γ^{NS} describe the weakest requirements to compile a source S into a binary B. That is, S can be successfully recompiled, into the same binary B, in every type environment which satisfies assumptions Γ^{NS}. Moreover, being those assumptions the *weakest* requirements, we do know that S cannot be compiled into B in every other environment which, conversely, does not satisfy Γ^{NS}.

This feature can be fruitfully used to implement a selective recompilation strategy, that is, a way to decide, after some sources have been changed, which *unchanged* sources have to be recompiled.

A recompilation strategy which does not guarantee the same outcome of an entire recompilation is not useful: why wasting time in debugging a program (a set of .class files in the Java case) which might behave *differently* from the program obtained recompiling all the sources from scratch?

Two contrasting requirements have to be considered: on the one hand recompilations can be rather expensive (in time), hence they should be avoided when possible. More precisely, they are useless when the recompilation of an unchanged (with respect to the previous compilation) source fragment S, whose corresponding binary fragment B is already present, would produce a binary equal to B. On the other hand, a recompilation strategy which saves time not recompiling a fragment `S`, with a corresponding binary fragment `B`, whose recompilation would produce a new binary `B′` different from `B`, could cost a lot of wasted time in debugging an *inconsistent* application, that is, an application that *cannot* be rebuilt by recompiling all the sources.

Albeit some Java IDEs support smart or incremental compilation, to our knowledge there are no publications which explain in detail the inner working of such recompilation strategies.

We say that a compilation strategy is *sound* if it is equivalent to a global recompilation; that is, if it recompiles all the changed sources and the unchanged sources whose new binary, produced by the overall recompilation, would differ from the existing one (if any) *and* all the unchanged sources for which the recompilation would fail. This latter requirement is very important: indeed, when the entire recompilation fails, so should do the partial recompilation.

Of course, a strategy which recompiles all the sources each time a change is made is trivially sound and, obviously, totally useless in practice. We say that a compilation strategy is *minimal* if it never recompiles an unchanged fragment whose new binary would be equal to the existing one.

The compilation strategy built on our type system is proved to be *sound* and *minimal* for a substantial subset of Java in Chapter 4.

1.5 Towards selective recompilation for full Java

Some Java-specific features, not modeled by the type system outlined in Section 1.4, hinder a straightforward implementation of the selective recompilation strategy described above to the full Java language. For this reason, in Chapter 4 we analyze these features and we extend the type system in order to take also into account:

- accessibility levels, that is, the fact that different clients have different visibility of the same class;
- different kinds of methods and fields, to reflect the fact that, e.g., an invocation to an instance method is translated into the JVM instruction `invokevirtual`, while an invocation of a static one into `invokestatic`;
- compile-time constant fields, because the bytecode generated for an access to such a field is the same that would be generated if the literal corresponding to the value of the field was used in its place;

- unreachable code detection, since the way Java handles unreachable code is peculiar: what is a warning in most languages is a compile-time error in Java, so we need to keep track of which code remains reachable after some sources have been changed.

With the exception of arrays and inner-classes, in the extended type system we model all the major features of Java: classes (including abstract classes), interfaces, primitive types, access modifiers (including packages, but without the `import` directive), constructors, (instance/static) fields (both in classes and interfaces), (instance/static/abstract) methods, super field accesses and method invocations, exceptions. The treatment of arrays and inner-classes would complicate the model without apparently giving further insights. In the few points where inner-classes would make a difference we briefly discuss the issue.

In order to provide a reasonable starting point for implementing a compilation manager, we also take in account some efficiency issues.

As said before, our selective recompilation strategy requires to check which assumptions, generated by a previous compilation, still hold in a new environment $\Gamma_{\texttt{new}}$ (that is, the environment extracted by the updated fragments) in order to determine which (unchanged) sources have to be recompiled. Because this implies to check whether $\Gamma_{\texttt{new}}$ is a well-formed environment, we analyze how well-formedness of environments can be verified incrementally.

This strategy is optimal, from a theoretical point of view, since the strategy triggers the recompilation of an unchanged source if and only if its recompilation produces a different binary (or an error). However, from a practical point of view, there is another point to ponder: the cost of checking whether the requirements of a fragment `S` are satisfied by a type environment $\Gamma^{\mathtt{S}}$. If this checking costed more than compiling the source `S`, then the whole idea would be useless.

In practice, what really matters is to have a recompilation strategy which is both *fast* and *sound*. Because in the global cost of recompilation we must take into account both the time spent in checks and the time spent in recompiling the selected sources, choosing the minimum number of sources to be recompiled does not necessarily mean choosing the fastest recompilation strategy. For these reasons, we consider a series of issues which should help in simplifying, and so speeding up, the checking step.

A compiler manager for full Java, based on these ideas, is under development.

Summary

Chapter 2 presents a framework for modeling Java-like separate compilation, introducing the notion of compilation schema and studying three different instantiations. In Chapter 3 we introduce an innovative type system where a fragment is typechecked in total isolation, w.r.t. a type environment consisting in fine-grained assumptions which describe the minimal requirements needed for generating a given binary. This system can be successfully exploited to define a sound recompilation strategy which never recompiles a source if not necessary. The extension of this strategy to full Java is analyzed in Chapter 4, where the type system is extended and implementation issues are discussed. Finally, in Chapter 5 we discuss related work, and in Chapter 6 we draw some conclusions and discuss further work.

Chapter 2

Java-like separate compilation

In this chapter we model the overall compilation process introducing the formal notion of *compilation schema*. A compilation schema consists of four components: two typing judgments, a type extraction function and a dependency function. The judgments are: a *source type judgment* $\Gamma^{\mathrm{s}} \vdash \mathtt{S} \leadsto \mathtt{B}$ and a *binary type judgment* $\Gamma^{\mathrm{s}} \vdash \mathtt{B} \diamond$ modeling typechecking, in a given standard[1] type environment Γ^{s}, of source code S and binary code B, respectively; in the former case the corresponding binary code B is also generated. These two components model the part of Java compilation which corresponds to truly separate compilation in the sense of [Car97]. The fact that in Java typechecking of a fragment may enforce typechecking of other fragments is modeled by the *dependency function*. Finally, the fact that in Java type information for a fragment cannot be provided separately from code is modeled by the *type extraction function* which extracts from a compilation environment *ce* (collection of source and binary fragments) a type environment Γ^{s} providing the type information needed for typechecking fragments in *ce*.

We consider three different compilation schemata for Java. The first, which we call *minimal*, corresponds to true separate typechecking, in the sense that no other fragment is typechecked when compilation is invoked on a fragment f. In this case, all inter-checks are left to the run-time verifier. However, note that some of the fragments f depends on *must* be available, since some of the type information needed for typechecking f has to be extracted from them[2].

The second compilation schema is the one used by standard Java compilers (at least for what we have been able to understand by experiments, since no specification is available). In this case only some inter-checks are performed: invoking compilation on f enforces

[1] We will introduce nonstandard type environments in Chapter 3.

[2] This is how standard Java compilers work; in Chapter 3 we drop this requirement by using the already mentioned nonstandard environments.

typechecking of other fragments, but not of all those which could be possibly loaded at run-time[3]; moreover no checks are performed on binary fragments, that is, the type judgment $\Gamma^{s} \vdash \mathtt{B}\diamond$ is always trivially valid. As a consequence, binaries obtained as result of the compilation are not guaranteed to safely link at run-time, as we show below. Finally, we propose a compilation schema which is type safe, that is, guarantees safe linking at run-time. For this last schema we provide a full definition of the four components for a small Java subset and we prove type safety.

2.1 Some motivating examples

In this section we illustrate, by means of some examples, three different Java compilation schemata, called *minimal*, *SDK* and *safe*, respectively.

As already explained, the minimal compilation schema requires the minimal amount of checks over fragments: typechecking is performed only for those source fragments on which the compiler has been explicitly invoked and no checks (except those strictly necessary for compiling the sources) are performed on binary fragments. This schema fits well in open environments where source fragments to be compiled are expected to be later dynamically linked with fragments that are not available at compile-time. For instance, assume that the class `C1` we want to compile depends on a class `C2`. Even in the case the source of `C2` is available, it could be sensible avoiding typechecking of `C2` if there is a high probability that it does not correspond to the actual code that will be linked with `C1` at run-time.

The SDK schema simply corresponds to the SDK implementation of Java[4]. As already said, this schema falls in between the minimal and the safe schema: it enforces more checks than the former but less than the latter. For instance, the compilation of a class `C` requires that all source fragments[5] directly used by `C` must be typechecked, while for all binary fragments[6] directly used by `C`, only their existence and format is checked (but no real typecheck is performed).

Finally, the safe schema can be sensibly applied when we expect that the fragments that will be linked at run-time are those available after the compilation; under this assumption, it makes sense to typecheck all fragments (either source or binary) used (either directly or indirectly) by a class `C`, in order to ensure that no execution of `C` will throw a linkage error (like, for instance, `NoClassDefFoundError` or `NoSuchMethodError`). To this aim, the compilation schema must include all those checks on binary fragments that a safe linker

[3]Note, however, that some fragment which are *not* loaded at run-time may be checked.

[4]These examples are based on version 1.4 beta 2 of SDK.

[5]Whose corresponding binary fragment is either unavailable or older.

[6]Which either are more recent than the corresponding source fragment or do not have a corresponding source fragment.

would perform if Java classes were statically linked[7].

While all these three schemata share the same source type judgment and type extraction function (corresponding to the Java type system defined in [GJSB00] and formalized in, e.g., [DE99]), they remarkably differ in the other two components (that is, dependency function and binary type judgment) as described in the following examples.

Consider the following class declarations, assuming that each one is contained in a single `.java` file:

```
class Main {
  static void main(String[] args) {
    new Used().m() ;
  }
  void g (UsedAsType x) {}
}

class Used extends UsedParent {
  int m() {
    return new TransUsed().m() ;
  }
}

class UsedParent{
  int m() {
    return 1 ;
  }
}

class TransUsed {
  int m() {
    return 1 ;
  }
}

class UsedAsType { ... }
```

As already stated, the same type extraction function is shared by the three schemata. The definition is straightforward: the type environment is a mapping associating with each available class `C` the type information which can be extracted from its code, that is, a

[7] Of course, such checks are usually performed at run-time by the JVM.

pair consisting of the direct superclass of `C` and the list of naked method headers[8] directly declared in `C`, which we will call the *class type* of `C`. A naked method header is a method header without the parameter names; roughly speaking it is the "signature" of the method, but this terminology would conflict with the one used by [GJSB00], where a method signature is just the pair consisting of the method name and parameter types. In the following we use the terms "naked method header" and "method header" interchangeably where there is no ambiguity.

Assume now that we want to compile `Main`. In the minimal schema, our aim is just to perform the separate typechecking (intra-checking), hence we only need the type information necessary to typecheck the source code of `Main`. In particular, for each class used in `Main`, we may need different information depending on the usage situation; for instance, when a class is used only as a type, like in the method `g`, we only need the existence of the class in the type environment, whereas when it is used as the type of the receiver in a method invocation, like in `new Used().m()`, we need to know which are *all* the method headers of the class (either directly declared or inherited). This information, which we will call the *full type* of a class, can be safely constructed by retrieving all the types of the ancestors of the class (provided that this hierarchy is acyclic); this will be formalized in Figure 2.7 later on. In summary, `Main.java` can be successfully typechecked, producing a corresponding `Main.class`, in the type environment Γ^{S} defined by

$$\{\texttt{Main} \mapsto \langle\texttt{Object}, \texttt{void g(UsedAsType)}\rangle, \texttt{Used} \mapsto \langle\texttt{UsedParent}, \texttt{int m()}\rangle,$$
$$\texttt{UsedParent} \mapsto \langle\texttt{Object}, \texttt{int m()}\rangle, \texttt{UsedAsType} \mapsto \ldots\},$$

which can be extracted from a compilation environment ce_1 which just contains the files `Main.java`, `Used.java`, `UsedParent.java` and `UsedAsType.java` (but not `TransUsed.java`) and no class files. This is formalized by the validity of the judgment $\Gamma^{\mathrm{S}} \vdash \texttt{S} \rightsquigarrow \texttt{B}$, defined in Figure 2.5. Note that, as already said, in Java type information on fragments cannot be provided separately from their code, so either the `.java` or `.class` files for `Used`, `UsedParent` and `UsedAsType` must be available, even though no typechecking is performed on their code (for `UsedAsType` not even the type information is used). In the minimal schema, indeed, the set of dependencies of `Main.java` is simply {`Main`}, reflecting the fact that we are only interested in typechecking `Main.java`.

In both the SDK and safe schema, the set of dependencies of `Main` in ce_1 includes also `Used`, `UsedParent`, `UsedAsType` and `TransUsed`. As a consequence, compilation of `Main.java` in ce_1 fails for both the SDK and safe schema, because ce_1 contains neither a source nor a binary file for `TransUsed`.

[8] In these examples we will consider for simplicity only instance method declarations, as in the Java subset defined in Section 2.3; in full Java the class type of a class includes also other declared members. We assume that the method `main` is just used for starting execution.

Let us consider now two compilation environments able to discriminate between the SDK and the safe schemata. First consider ce_2 which contains the source files `Main.java`, `Used.class`, `UsedParent.class`, `UsedAsType.class`[9], and a changed version of `TransUsed.java` which does not satisfy intra-checks (for instance, the body of method `m` could return a boolean). The type environment extracted from ce_2 is still Γ^{s} but, the set of dependencies of `Main` in ce_2 is now $\{\mathtt{Main}, \mathtt{Used}, \mathtt{UsedParent}, \mathtt{UsedAsType}, \mathtt{TransUsed}\}$ in the safe schema, and $\{\mathtt{Main}, \mathtt{Used}, \mathtt{UsedParent}, \mathtt{UsedAsType}\}$ in the SDK schema.

Hence, in SDK, although the source of class `TransUsed` is not typechecked, a new binary fragment `Main.class` is produced anyway. However, in the new environment obtained by enriching ce_2 with the fragment `Main.class`, the execution of class `Main` throws the error `NoClassDefFoundError` (note that this error is raised instead of a type error since the class has not been compiled, hence there is no corresponding binary), whereas this error is detected at compile-time by the safe schema which performs the typechecking of the source of `TransUsed`.

In this case the difference between the Java and the safe schema is given by the dependency function. However, even in the case dependencies are the same, the two schemata can still behave differently due to the fact that the safe schema also performs a significant binary typechecking (formalized by the validity of the judgment $\Gamma^{\mathrm{s}} \vdash \mathtt{B} \diamond$ that will be defined in Figure 2.8). For instance, invoking the compilation of both `Main` and `TransUsed` in the compilation environment ce_3 which contains `Main.java`, `Used.class`, `UsedParent.class`, `UsedAsType.class` and a changed version of `TransUsed.java` which does not satisfy type requirements in `Used` (for instance, declaring `boolean m() {return true;}`), in SDK no checks are performed on the the binary code of `Used`. Hence, again, in the binary environment obtained after the compilation, the execution of class `Main` throws the exception `NoSuchMethodError`, whereas this error is detected at compile-time by the safe schema.

2.2 Framework

We now formally define our framework for modeling the Java overall compilation process. Consistently with this aim, we use a Java-related terminology everywhere. However, most of the notions presented here could be generalized to model the compilation process of other languages.

Let us denote by $\mathbb{C}$ the set of fragment names, that is, in Java, class/interface names[10], ranged over by `C`, and by $\mathbb{S}$ and $\mathbb{B}$ the set of source and binary fragments, respectively. We assume that source fragments are `.java` files containing (for simplicity) exactly one

[9] Obtained, e.g., by compiling the whole program in the example.

[10] We will consider only classes in the Java subsets of the first chapters. We use a larger subset only in Chapter 4.

class/interface declaration, and binary fragments are `.class` files.

We denote by $[A \rightharpoonup_{\mathit{fin}} B]$ the set of *finite partial functions* from A into B, that is, functions f from A into B which are defined on a finite subset of A, denoted $\mathit{Def}(f)$.

So, a *compilation environment* ce is a pair

$$\langle ce_b, ce_s \rangle \in CE = [\mathbb{C} \rightharpoonup_{\mathit{fin}} \mathbb{B}] \times [\mathbb{C} \rightharpoonup_{\mathit{fin}} \mathbb{S}]$$

s.t. $\mathit{Def}(ce_b) \cap \mathit{Def}(ce_s) = \emptyset$. We will call ce_b and ce_s a *binary* and a *source environment*, respectively. Note that the assumption $\mathit{Def}(ce_b) \cap \mathit{Def}(ce_s) = \emptyset$ means that, even in the case a class has both a binary and a source definition, the compiler considers only one of them, according to some rule. Here we do not specify any rule, and it can be assumed the rule simply consists in considering the latest modified one (according to the timestamps of the corresponding files). Indeed, this is what happens in most implementations, albeit this is not always satisfactory as we discuss in Section 4.2.2. The results of (successful) compilations are binary environments. Hence, we can model the compilation process by a (partial) function, called *compilation function*:

$$\mathcal{C} : CE \times \wp(\mathbb{C}) \rightharpoonup [\mathbb{C} \rightharpoonup_{\mathit{fin}} \mathbb{B}]$$

where $\mathcal{C}(\langle ce_b, ce_s \rangle, \mathtt{CS}) = ce'_b$ intuitively means that the compilation, invoked on fragments with names in $\mathtt{CS}$, in the compilation environment consisting of binary fragments ce_b and source fragments ce_s, generates binary fragments ce'_b.

We introduce now the formal notion of *compilation schema*, meant to express different Java compilation processes.

A compilation schema consists of the following four components.

- A *dependency function* $\mathcal{D}$ which gives, for any compilation environment ce and set of fragment names $\mathtt{CS}$, the set $\mathtt{CS}'$ of all the fragment names on which typechecking is enforced when the compilation is invoked on $\mathtt{CS}$.
- A *type extraction function* $\mathcal{T}$ which extracts from a compilation environment ce a *type environment* Γ^{S} providing the type information necessary to typecheck fragments in ce.
- A *source type judgment* $\Gamma^{\mathrm{S}} \vdash \mathtt{S} \leadsto \mathtt{B}$ expressing that in the type environment Γ^{S} the source fragment $\mathtt{S}$ is successfully typechecked generating the binary fragment $\mathtt{B}$.
- A *binary type judgment* $\Gamma^{\mathrm{S}} \vdash \mathtt{B} \diamond$ expressing that in the type environment Γ^{S} the binary fragment $\mathtt{B}$ is successfully typechecked.

$$\dfrac{\forall \mathtt{C} \in \mathtt{CS}_b\ \Gamma^{\mathrm{s}} \vdash ce_b(\mathtt{C}) \diamond \qquad \forall \mathtt{C} \in \mathtt{CS}_s\ \Gamma^{\mathrm{s}} \vdash ce_s(\mathtt{C}) \leadsto \mathtt{B}_{\mathtt{C}}}{\mathcal{C}(\langle ce_b, ce_s\rangle, \mathtt{CS}) = \{\mathtt{C} \mapsto \mathtt{B}_{\mathtt{C}} \mid \mathtt{C} \in \mathtt{CS}_s\}} \quad \begin{array}{l} \Gamma^{\mathrm{s}} = \mathcal{T}(\langle ce_b, ce_s\rangle) \\ \mathtt{CS} \subseteq \mathit{Def}(ce_s) \\ \mathtt{CS}_d = \mathcal{D}(\langle ce_b, ce_s\rangle, \mathtt{CS}) \\ \mathtt{CS}_b = \mathtt{CS}_d \cap \mathit{Def}(ce_b) \\ \mathtt{CS}_s = \mathtt{CS}_d \cap \mathit{Def}(ce_s) \end{array}$$

Figure 2.1: Definition of compilation function

$$\forall \mathtt{C}\ \mathcal{T}(\langle ce_b, ce_s\rangle)(\mathtt{C}) = \begin{cases} \langle \mathtt{C}', \mathcal{T}(\mathtt{MDS}^b)\rangle & \text{if } ce_b(\mathtt{C}) = \langle \mathtt{C}, \mathtt{C}', \mathtt{MDS}^b, \mathtt{E}^b\rangle \\ \langle \mathtt{C}', \mathcal{T}(\mathtt{MDS}^s)\rangle & \text{if } ce_s(\mathtt{C}) = \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \}\ \mathtt{main\ E}^s \\ \bot & \text{otherwise} \end{cases}$$
$$\mathcal{T}(\mathtt{MD}^s_1 \ldots\ \mathtt{MD}^s_n) = \mathcal{T}(\mathtt{MD}^s_1) \ldots\ \mathcal{T}(\mathtt{MD}^s_n)$$
$$\mathcal{T}(\mathtt{MD}^b_1 \ldots\ \mathtt{MD}^b_n) = \mathcal{T}(\mathtt{MD}^b_1) \ldots\ \mathcal{T}(\mathtt{MD}^b_n)$$
$$\mathcal{T}(\mathtt{MH}\ \{\ \mathtt{return\ E}^s;\ \}) = \mathcal{T}(\mathtt{MH})$$
$$\mathcal{T}(\mathtt{MH}\ \{\ \mathtt{return\ E}^b;\ \}) = \mathcal{T}(\mathtt{MH})$$
$$\mathcal{T}(\mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)) = \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1 \ldots \mathtt{T}_n)$$

Figure 2.2: Definition of the type extraction function

To compile a set of fragments $\mathtt{CS}$ in a compilation environment ce, first the needed type environment Γ^{s} is extracted applying $\mathcal{T}$ to ce. Then, all the fragments in the set $\mathtt{CS}_d$ computed from $\mathtt{CS}$ using $\mathcal{D}$ are typechecked generating corresponding binaries for those which were in source form. This can be formalized by the inference rule in Figure 2.1 which defines a compilation function $\mathcal{C}$ in terms of the four components of a compilation schema. The second side condition, $\mathtt{CS} \subseteq \mathit{Def}(ce_s)$, states that compilations can be only invoked on a set of existing sources.

Let us now apply the above definitions for specifying the three different compilation schemata informally introduced in Section 2.1.

The type extraction function is the same for the three schemata: the type environment extracted from a compilation environment $ce = \langle ce_b, ce_s\rangle$ is a finite partial function which associates to each $\mathtt{C} \in \mathit{Def}(ce_b) \cup \mathit{Def}(ce_s)$ its *class type*, that is, a pair consisting of the superclass of $\mathtt{C}$ and the list of the method headers declared in $\mathtt{C}$, as shown in Figure 2.2.

The source type judgment is the same for the three schemata as well, and corresponds to the Java type system defined in [GJSB00] and formalized, e.g., in [DE99]. The formalization for the small Java subset for which we define a type safe compilation schema is given in Figure 2.5.

Despite the first two components are the same, the three schemata remarkably differ in the remaining components.

For what concerns the dependency function, $\mathcal{D}(ce, \{\texttt{C}\})$ contains only C in the minimal schema; in the safe schema $\mathcal{D}(ce, \{\texttt{C}\})$ contains C and all the classes transitively used by C, regardless that C is in source or binary form (see the formal definition in Figure 2.4 later on). In the SDK schema the definition is much more involved. First of all, $\mathcal{D}(ce, \{\texttt{C}\})$ contains C and all the classes directly used by C. For each of these classes, say C′, $\mathcal{D}(ce, \{\texttt{C}\})$ also (recursively) contains $\mathcal{D}(ce, \{\texttt{C}'\})$ if C′ is in source form. If C′ is in binary form, then the behavior is different depending whether C′ is only used in C as "abstract" type (for instance, field type, parameter type, method return type) or information on the components provided by C is also needed (for instance, there is a method invocation with the type of the receiver C). In the former case $\mathcal{D}(ce, \{\texttt{C}\})$ contains only C′ and some of its ancestors, in the latter it contains C′, all the ancestor classes of C′ and (recursively) $\mathcal{D}(ce, \{\texttt{C}''\})$ for each ancestor C″ which is in source form. This rule is quite complex, and has been extrapolated by performing a number of compilation tests because no form of documentation seems to be available.

Also, the binary type judgment differs from schema to schema. In the minimal schema no typechecks are performed (that is, the judgment $\Gamma^{\mathrm{s}} \vdash \texttt{B} \diamond$ trivially holds). In the safe schema the checks performed on a binary fragment are similar to those performed on a source fragment. A difference is, for instance, the way a method invocation is checked. In the source case the method must be found searching in all the ancestor classes, and overloading must be resolved, while in the binary case a method invocation is already annotated with the class where the method should be found together with its header. The formalization for the small Java subset for which we define a type safe compilation schema is given in Figure 2.5 (last rule) and Figure 2.6 for the source case and in Figure 2.8 (last rule) for the binary case.

In the SDK schema no typechecks are performed on binary code. The only checks which are performed, when typechecking a class C which uses C′, together with the existence check on C′, are the existence, and the correctness of the format, of the binary (analogous to the fact that Java grammar is respected in the source case) and on the correspondence between the fragment name and the name of the class defined inside. For simplicity, in the formal model in Section 2.3 we assume that fragments are well-formed in this sense.

We introduce now the formal property of type safety for separate compilation. We assume a judgment of the form $\texttt{C} \leadsto_{ce_b} V$ which is valid if and only if execution of class C in the binary environment ce_b terminates producing a value V which can be either a normal value or a linkage exception. Intuitively, this judgment corresponds to start the execution from class C in an environment ce_b corresponding to the set of all available binaries that can be dynamically linked during the execution.

The formal definition of this judgment for the small Java subset for which we define a type safe compilation schema is given in Figure 2.15. Let us denote with $ce_b[ce'_b]$ the partial function f s.t. $Def(f) = Def(ce_b) \cup Def(ce'_b)$ and for any $\mathtt{C} \in Def(f)$ $f(\mathtt{C}) = ce'_b(\mathtt{C})$ if $\mathtt{C} \in Def(ce'_b)$ and $f(\mathtt{C}) = ce_b(\mathtt{C})$ otherwise.

Def. 2.2.1 *A compilation function* $\mathcal{C}$ *is* type safe *iff for any compilation environment* $\langle ce_b, ce_s\rangle$ *and set of class names* CS*, if* $\mathcal{C}(\mathit{CS}, \langle ce_b, ce_s\rangle) = ce'_b$*, then, for all class names* $\mathtt{C} \in Def(ce'_b)$ *and values* V*, if* $\mathtt{C} \leadsto_{ce_b[ce'_b]} V$*, then* V *is not a (linkage) exception.*

Note that type safety requires that execution does not throw linkage errors only when started from classes that are the product of the compilation. Indeed, an error raised by an execution started from a class C present in the original binary environment ce_b can be either an error which was already present, hence not due to the compilation, or an error due to the fact that some binary used by C has been modified.

2.3 A safe compilation schema

The language we consider is reminiscent of Featherweight Java [IPW99], in the sense that it is a small functional subset of Java (see Figure 2.3); however, since here we are mainly interested in code generation and bytecode execution, we present a simple binary language as well, together with its reduction semantics. The dynamic semantics of our Java subset is indirectly defined by a compilation function mapping well-typed source fragments into well-typed binary fragments.

Metavariables C, m, x and N range over sets of class, method and parameter names, and integer literals, respectively.

A source fragment S is a class declaration consisting of the class name, the name of the superclass, a sequence of method declarations $\mathtt{MDS}^s$, and an expression $\mathtt{E}^s$ playing the role of the (static) `main` method, that for simplicity we assume present in all classes. A method declaration $\mathtt{MD}^s$ consists of a method header and a method body (an expression). A method header MH consists of a (return) type, a method name and a sequence of parameter types and names. There are four kinds of expression: instance creation[11], parameter name, integer literal and method invocation. A type is either a class name or `int`. We will use the abbreviation $\bar{\mathtt{T}}$ for $\mathtt{T}_1 \ldots \mathtt{T}_n$ in the following.

Our description of bytecode is rather abstract: we basically enrich the source language with two kinds of annotation. Each method invocation is annotated with a method descriptor, which describes the method which has been statically selected for the invocation. The

[11]Although we do not model constructors in this subset of Java, we chose to put "()" after the class name anyway, to mimic the syntax Java programmers are used to.

$$
\begin{array}{rcl}
\texttt{S} & ::= & \texttt{class C extends C}' \; \{ \; \texttt{MDS}^s \; \} \; \texttt{main E}^s \\
\texttt{MDS}^s & ::= & \texttt{MD}^s_1 \ldots \; \texttt{MD}^s_n \\
\texttt{MD}^s & ::= & \texttt{MH} \; \{ \; \texttt{return E}^s; \; \} \\
\texttt{MH} & ::= & \texttt{T}_0 \; \texttt{m}(\texttt{T}_1 \; \texttt{x}_1, \ldots, \texttt{T}_n \; \texttt{x}_n) \\
\texttt{E}^s & ::= & \texttt{new C}() \mid \texttt{x} \mid \texttt{N} \mid \\
 & & \texttt{E}^s_0.\texttt{m}(\texttt{E}^s_1, \ldots, \texttt{E}^s_n) \\
 & & \\
\texttt{T} & ::= & \texttt{C} \mid \texttt{int} \\
 & & \\
\texttt{B} & ::= & \langle \texttt{C}, \texttt{C}', \texttt{MDS}^b, \texttt{E}^b \rangle \\
\texttt{MDS}^b & ::= & \texttt{MD}^b_1 \ldots \; \texttt{MD}^b_n \\
\texttt{MD}^b & ::= & \texttt{MH} \; \{ \; \texttt{return E}^b; \; \} \\
V & ::= & \texttt{new C}() \mid \texttt{N} \mid \varepsilon \\
\texttt{E}^b & ::= & V \mid \texttt{x} \mid \texttt{E}^b_0.\texttt{m} \ll \boxed{\texttt{C}}.\texttt{T} \; (\texttt{T}_1 \ldots \texttt{T}_n) \gg (\texttt{E}^b_1, \ldots, \texttt{E}^b_n) \\
 & & \texttt{E}^b_0.\texttt{m} \ll \texttt{C.T} \; (\texttt{T}_1 \ldots \texttt{T}_n) \gg (\texttt{E}^b_1, \ldots, \texttt{E}^b_n) \mid \texttt{new} \; \boxed{\texttt{C}}() \\
\varepsilon & ::= & \mathit{ClassNotFound} \mid \mathit{ClassCircularityError} \mid \mathit{VerifyError} \mid \mathit{NoSuchMethod} \\
 & & \\
\texttt{NH} & ::= & \texttt{T} \; \texttt{m}(\texttt{T}_1 \ldots \texttt{T}_n) \\
\texttt{ANH} & ::= & \texttt{C NH} \\
\texttt{ANHS} & ::= & \texttt{ANH}_1 \ldots \; \texttt{ANH}_n
\end{array}
$$

Figure 2.3: Syntax and types

descriptor consists of: the static type of the receiver[12], the return type and the type of the parameters. Moreover, each class name mentioned either in class creation or as the first component of method descriptors in method invocation is (initially) boxed[13]. The idea is that a reference to a class is "sealed" in a box until it has been verified (at run-time). Such a reference cannot be used until it is unboxed.

A binary fragment $\mathtt{B}$ consists of the name of the class, the name of the superclass, a set of binary method declarations $\mathtt{MDS}^b$ and the binary expression corresponding to the method `main`. This expression is used as the entry point of the program when the class is executed. A binary method declaration $\mathtt{MD}^b$ is structurally equivalent to a source method declaration except that the body is a binary expression.

Binary expressions can be either *values*, or parameters, or (either boxed or unboxed) method invocations, or a boxed creation expression. Values correspond to the normal forms of the reduction semantics of binary fragments (defined in Figure 2.15), and can be either unboxed creation expressions, or integer literals, or exceptions (in case of abnormal termination).

Note that exceptions and unboxed method invocation and creation are only needed for defining the rewriting rules for bytecode execution (see Figure 2.15), but they are not considered valid binary formats, even though for sake of simplicity we do not have introduced two separate syntactic categories corresponding to valid binary format and valid run-time expressions, respectively.

In the last part of Figure 2.3 we define *naked (method) headers* and *annotated naked (method) headers*, which are not part of the syntax but will be used in the type judgments. A naked method header $\mathtt{NH}$ is a method header without the argument names; an annotated naked method header $\mathtt{ANH}$ is a naked (method) header prefixed by an annotation indicating the class which contains the method declaration.

We start now the formal definition of the four components of our safe compilation schema, which are used (as shown in Figure 2.1 in Section 2.2) to define the corresponding compilation function.

The dependency function $\mathcal{D}$ is defined as follows:

$$\mathcal{D}(ce, \mathtt{CS}) = \{\mathtt{C}' \mid \exists \mathtt{C} \in \mathtt{CS}\ s.t.\ \mathtt{C} \xrightarrow{*}_{ce} \mathtt{C}'\}$$

where $\xrightarrow{*}_{ce}$ is the reflexive and transitive closure of the relation $\rightarrow_{ce}$ defined by $\mathtt{C} \rightarrow_{ce} \mathtt{C}'$ iff $\mathtt{C}' \in \mathit{refClasses}(\mathtt{C}, ce)$. This latter function, defined in Figure 2.4, gives the set of all classes

[12]This is a change introduced in SDK 1.4, since in the previous versions the first component of method descriptors corresponded to the class where the method was statically found.

[13]This notion has *nothing* to do with boxing/unboxing of C#. Ours is a way to denote an unverified item at run-time, while the boxing of C# is a compiler feature which, roughly speaking, let the programmer use primitive and reference type interchangeably.

$$
\begin{array}{l}
\mathit{refClasses}(\mathtt{C}, \langle ce_b, ce_s\rangle) = \begin{cases} \mathit{refClasses}(ce_b(\mathtt{C})) & \text{if } \mathtt{C} \in \mathit{Def}(ce_b) \\ \mathit{refClasses}(ce_s(\mathtt{C})) & \text{if } \mathtt{C} \in \mathit{Def}(ce_s) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{refClasses}(\mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \}\ \mathtt{main\ E}^s) = \\
\quad \{\mathtt{C}, \mathtt{C'}\} \cup \mathit{refClasses}(\mathtt{MDS}^s) \cup \mathit{refClasses}(\mathtt{E}^s) \\
\mathit{refClasses}(\mathtt{MD}^s_1 \ldots\ \mathtt{MD}^s_n) = \bigcup_{i\in 1..n} \mathit{refClasses}(\mathtt{MD}^s_i) \\
\mathit{refClasses}(\mathtt{MH}\ \{\ \mathtt{return\ E}^s;\ \}) = \mathit{refClasses}(\mathtt{MH}) \cup \mathit{refClasses}(\mathtt{E}^s) \\
\mathit{refClasses}(\mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)) = \{\mathtt{T}_0, \ldots, \mathtt{T}_n\} \\
\mathit{refClasses}(\mathtt{new\ C}()) = \{\mathtt{C}\} \\
\mathit{refClasses}(\mathtt{x}) = \mathit{refClasses}(\mathtt{N}) = \emptyset \\
\mathit{refClasses}(\mathtt{E}^s_0.\mathtt{m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n)) = \bigcup_{i\in 0..n} \mathit{refClasses}(\mathtt{E}^s_i) \\
\\
\mathit{refClasses}(\langle \mathtt{C}, \mathtt{C'}, \mathtt{MDS}^b, \mathtt{E}^b\rangle) = \{\mathtt{C}, \mathtt{C'}\} \cup \mathit{refClasses}(\mathtt{MDS}^b) \cup \mathit{refClasses}(\mathtt{E}^b) \\
\mathit{refClasses}(\mathtt{MD}^b_1 \ldots\ \mathtt{MD}^b_n) = \bigcup_{i\in 1..n} \mathit{refClasses}(\mathtt{MD}^b_i) \\
\mathit{refClasses}(\mathtt{MH}\ \{\ \mathtt{return\ E}^b;\ \}) = \mathit{refClasses}(\mathtt{MH}) \cup \mathit{refClasses}(\mathtt{E}^b) \\
\mathit{refClasses}(\mathtt{E}^b_0.\mathtt{m} \ll \boxed{\mathtt{C}}.\mathtt{T}_0\ (\mathtt{T}_1 \ldots \mathtt{T}_n) \gg (\mathtt{E}^b_1, \ldots, \mathtt{E}^b_n)) = \{\mathtt{C}\} \cup \bigcup_{i\in 0..n} (\mathit{refClasses}(\mathtt{E}^b_i) \cup \{\mathtt{T}_i\}) \\
\mathit{refClasses}(\mathtt{new}\ \boxed{\mathtt{C}}()) = \{\mathtt{C}\}
\end{array}
$$

Figure 2.4: Definition of function *refClasses*

explicitly mentioned in the code of C.

A standard[14] type environment Γ^{s} is a finite (partial) function from class names into *class types* which are pairs $\langle \mathtt{C'}, \mathtt{NH}_1 \ldots \mathtt{NH}_n\rangle$ where $\mathtt{C'}$ is the (direct) superclass of C and $\mathtt{NH}_1 \ldots \mathtt{NH}_n$ is the list of the headers of methods declared in C.

The type extraction function $\mathcal{T}$, defined in Figure 2.2 (page 24), simply throws away method bodies and parameter names, retaining type information from all classes in the compilation environment.

Figure 2.5 shows the rules for typechecking, in a given type environment, source fragments (with generation of the corresponding binary code).

The first rule defines the typechecking of a class declaration.

First, it is checked that C has a well-formed *full type* in Γ^{s}. The full type of a class is the list of all the annotated method headers of the class, either directly declared or inherited, and it can be safely constructed if there are no cycles in the inheritance hierarchy of C and the Java rules on method overriding are respected. This is formalized by the judgment $\Gamma^{\mathrm{s}} \vdash \mathtt{C} ::_\diamond \_$ defined in Figure 2.7. When we are not interested in specifying a particular

[14]As said before, we introduces nonstandard type environments in Chapter 3.

$$\frac{\Gamma^{\mathrm{s}} \vdash \mathtt{C} ::_{\diamond} \_ \quad \Gamma^{\mathrm{s}} \vdash \mathtt{MDS}^s \leadsto \mathtt{MDS}^b \quad \Gamma^{\mathrm{s}}; \emptyset \vdash \mathtt{E}^s : \_ \leadsto \mathtt{E}^b}{\Gamma^{\mathrm{s}} \vdash \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \}\ \mathtt{main}\ \mathtt{E}^s \leadsto \langle \mathtt{C}, \mathtt{C'}, \mathtt{MDS}^b, \mathtt{E}^b \rangle}$$

$$\frac{\forall i \in 1..n\ \Gamma^{\mathrm{s}} \vdash \mathtt{MD}_i^s \leadsto \mathtt{MD}_i^b}{\Gamma^{\mathrm{s}} \vdash \mathtt{MD}_1^s\ \ldots\ \mathtt{MD}_n^s \leadsto \mathtt{MD}_1^b \ldots \mathtt{MD}_n^b}$$

$$\frac{\begin{array}{l}\Gamma^{\mathrm{s}}; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E}^s : \mathtt{T} \leadsto \mathtt{E}^b \\ \Gamma^{\mathrm{s}} \vdash \mathtt{T} \leq \mathtt{T}_0\end{array}}{\begin{array}{l}\Gamma^{\mathrm{s}} \vdash\ \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E}^s;\ \} \leadsto \\ \qquad \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E}^b;\ \}\end{array}}\ \forall i \in 1..n\ \mathtt{T}_i \in \mathit{Def}(\Gamma^{\mathrm{s}}) \cup \{\mathtt{int}\}$$

$$\frac{}{\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{new\ C()} : \mathtt{C} \leadsto \mathtt{new}\ \boxed{\mathtt{C}}\,()}\ \mathtt{C} \in \mathit{Def}(\Gamma^{\mathrm{s}})$$

$$\frac{}{\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{N} : \mathtt{int} \leadsto \mathtt{N}}$$

$$\frac{}{\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{x} : \mathtt{T} \leadsto \mathtt{x}}\ \Pi(\mathtt{x}) = \mathtt{T}$$

$$\frac{\begin{array}{l}\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{E}_0^s : \mathtt{C} \leadsto \mathtt{E}_0^b \\ \forall i \in 1..n\ \Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{E}_i^s : \mathtt{T}_i \leadsto \mathtt{E}_i^b\end{array}}{\begin{array}{l}\Gamma^{\mathrm{s}}; \Pi \vdash\ \mathtt{E}_0^s.m(\mathtt{E}_1^s, \ldots, \mathtt{E}_n^s) : \mathtt{T'} \leadsto \\ \qquad \mathtt{E}_0^b.m \ll \boxed{\mathtt{C}}.\mathtt{T'}\ (\bar{\mathtt{T}}') \gg (\mathtt{E}_1^b, \ldots, \mathtt{E}_n^b)\end{array}}\ <\bar{\mathtt{T}}', \mathtt{T'}> = \mathit{methRes}(\Gamma^{\mathrm{s}}, \mathtt{C}, m, \mathtt{T}_1 \ldots \mathtt{T}_n)$$

Figure 2.5: Source type-judgment

value/metavariable, we use the wildcard notation _ in its place.

Then, the method bodies and the main expression are checked and compiled.

The second rule defines the typechecking of a sequence of method declarations. Each method declaration is correct (third rule) if the type of the expression body is a subtype of the declared return type (premises) and all argument types exist in Γ^{s} (note that *no* check is performed on the fact that they have well-formed full types). The judgment $\Gamma^{\mathrm{s}} \vdash \mathtt{T} \leq \mathtt{T}'$ is valid whenever $\mathtt{T}$ is a subtype of $\mathtt{T}'$ in the type environment Γ^{s}, and is defined in Figure 2.9.

Other rules define the typechecking of expressions, which also needs a local type environment Π which is a (partial) function from parameters into types.

An instance creation expression, `new C()`, is well-typed, and has type `C`, in Γ^{s} and Π if `C` exists in Γ^{s}. An integer literal is trivially well-typed, and has type `int`, in every Γ^{s} and Π.

A parameter is well-typed in Γ^{s} and Π if it belongs to the domain of the local type environment, and it has the corresponding type.

A method invocation expression is typechecked in two steps: first, the receiver expression and all the argument expressions are typechecked finding their types $\mathtt{T}_i$ (and generating the corresponding binary expressions $\mathtt{E}^b_i$). Then, using this information, the most specific among the applicable methods is selected, as formally defined by the function *methRes*, defined in Figure 2.6, which returns a pair consisting of the type of parameters and returned value used to annotate the binary method invocation produced as the result of the compilation; the annotation is used at run-time by the JVM (see Figure 2.15). Recall that since SDK 1.4 the first component of the descriptor, which annotates the method invocation, is the static type of the receiver (`C` in the rule).

In Figure 2.7 we define the judgment $\Gamma^{\mathrm{s}} \vdash \mathtt{C} :: \mathtt{ANHS}$, associating with a class its full type `ANHS`, and the judgment $\Gamma^{\mathrm{s}} \vdash \mathtt{C} ::_{\diamond} \mathtt{ANHS}$, which is valid only if `ANHS` is well-formed.

As already said, the full type of a class consists of the sequence of the annotated headers of the methods either directly declared in `C` or inherited.

A full class type `ANHS` is well-formed if it does not contain duplicate method headers and if the Java rules on overriding are satisfied (predicate *okOverride*).

In Figure 2.7 the notation $\mathtt{ANH} \in \mathtt{ANHS}$ is a shortcut for $\exists \mathtt{ANHS}_0, \mathtt{ANHS}_1 : \mathtt{ANHS} = \mathtt{ANHS}_0\ \mathtt{ANH}\ \mathtt{ANHS}_1$.

Note that neither $\Gamma^{\mathrm{s}} \vdash \mathtt{C} :: \mathtt{ANHS}$ nor $\Gamma^{\mathrm{s}} \vdash \mathtt{C} ::_{\diamond} \mathtt{ANHS}$ can be deduced for `C` if it has a cyclic inheritance hierarchy in Γ^{s}.

Figure 2.8 shows the rules for typechecking binary fragments, which are analogous to those for source fragments shown in Figure 2.5, except for the last rule, concerning method invocations.

$$methRes(\Gamma^{\mathrm{s}}, \mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \begin{cases} {<}\bar{\mathtt{T}}', \mathtt{T}'{>} & \text{if } \Gamma^{\mathrm{s}} \vdash \mathtt{C} :: \mathtt{ANHS} \wedge \\ & \quad mostSpec(\Gamma^{\mathrm{s}}, appMeth(\Gamma^{\mathrm{s}}, \mathtt{ANHS}, \mathtt{m}, \bar{\mathtt{T}})) = \mathtt{C}'\ \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}') \\ \perp & \text{otherwise} \end{cases}$$

$$appMeth(\Gamma^{\mathrm{s}}, \Lambda, \mathtt{m}, \bar{\mathtt{T}}) = \Lambda$$
$$appMeth(\Gamma^{\mathrm{s}}, \mathtt{C}\ \mathtt{T}\ \mathtt{m}'(\bar{\mathtt{T}}')\ \mathtt{ANHS}, \mathtt{m}, \bar{\mathtt{T}}) =$$
$$\begin{cases} \mathtt{C}\ \mathtt{T}\ \mathtt{m}'(\bar{\mathtt{T}}')\ appMeth(\Gamma^{\mathrm{s}}, \mathtt{ANHS}, \mathtt{m}, \bar{\mathtt{T}}) & \text{if } \mathtt{m} = \mathtt{m}' \wedge \Gamma^{\mathrm{s}} \vdash \bar{\mathtt{T}} \leq \bar{\mathtt{T}}' \\ appMeth(\Gamma^{\mathrm{s}}, \mathtt{ANHS}, \mathtt{m}, \bar{\mathtt{T}}) & \text{otherwise} \end{cases}$$

$$mostSpec(\Gamma^{\mathrm{s}}, \Lambda) = \perp$$
$$mostSpec(\Gamma^{\mathrm{s}}, \mathtt{ANHS}) = \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \text{ if } \begin{cases} \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{ANHS} \text{ and,} \\ \forall\ \mathtt{C}'\ \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}') \in \mathtt{ANHS} : \Gamma^{\mathrm{s}} \vdash \mathtt{C}\ \bar{\mathtt{T}} \leq \mathtt{C}'\ \bar{\mathtt{T}}' \end{cases}$$

Figure 2.6: Definition of *methRes*

$$\frac{}{\Gamma^{\mathrm{s}} \vdash \mathtt{Object} :: \Lambda}$$

$$\frac{\Gamma^{\mathrm{s}} \vdash \mathtt{C}' :: \mathtt{ANHS}'}{\Gamma^{\mathrm{s}} \vdash \mathtt{C} :: \mathtt{ANHS}'\ \mathtt{C}\ \mathtt{NH}_1\ \ldots\ \mathtt{C}\ \mathtt{NH}_n} \quad \begin{array}{l} \Gamma^{\mathrm{s}}(\mathtt{C}) = \langle \mathtt{C}', \mathtt{NH}_1 \ldots \mathtt{NH}_n \rangle \\ \mathtt{NH}_i = \mathtt{NH}_j \implies i = j \end{array}$$

$$\frac{\Gamma^{\mathrm{s}} \vdash \mathtt{C} :: \mathtt{ANHS}}{\Gamma^{\mathrm{s}} \vdash \mathtt{C} ::_{\diamond} \mathtt{ANHS}}\ okOverride(\mathtt{ANHS})$$

$$okOverride(\mathtt{ANHS}) \iff$$
$$\forall \mathtt{ANH}, \mathtt{ANH}' \in \mathtt{ANHS}\ \mathtt{ANH} = \mathtt{C}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \wedge \mathtt{ANH}' = \mathtt{C}'\ \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \implies \mathtt{T} = \mathtt{T}'$$

Figure 2.7: Full type of a class

$$\frac{\Gamma^{\mathrm{s}} \vdash \mathtt{C} ::_{\diamond} \_ \quad \Gamma^{\mathrm{s}} \vdash \mathtt{MDS}^b \diamond \quad \Gamma^{\mathrm{s}}; \emptyset \vdash \mathtt{E}^b : \_ \diamond}{\Gamma^{\mathrm{s}} \vdash \langle \mathtt{C}, \_, \mathtt{MDS}^b, \mathtt{E}^b \rangle \diamond}$$

$$\frac{\forall i \in 1..n \ \Gamma^{\mathrm{s}} \vdash \mathtt{MD}_i^b \diamond}{\Gamma^{\mathrm{s}} \vdash \mathtt{MD}_1^b \ \ldots \ \mathtt{MD}_n^b \diamond}$$

$$\frac{\begin{array}{l}\Gamma^{\mathrm{s}}; \{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E}^b : \mathtt{T} \diamond \\ \Gamma^{\mathrm{s}} \vdash \mathtt{T} \leq \mathtt{T}_0\end{array}}{\Gamma^{\mathrm{s}} \vdash \mathtt{T}_0 \ \mathtt{m}(\mathtt{T}_1 \ \mathtt{x}_1, \ldots, \mathtt{T}_n \ \mathtt{x}_n) \ \{ \ \mathtt{return} \ \mathtt{E}^b; \ \} \diamond} \ \forall i \in 1..n \ \mathtt{T}_i \in \mathit{Def}(\Gamma^{\mathrm{s}}) \cup \{\mathtt{int}\}$$

$$\frac{}{\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{new} \ \boxed{\mathtt{C}} \, () : \mathtt{C} \diamond} \ \mathtt{C} \in \mathit{Def}(\Gamma^{\mathrm{s}})$$

$$\frac{}{\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{N} : \mathtt{int} \diamond}$$

$$\frac{}{\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{x} : \mathtt{T} \diamond} \ \Pi(\mathtt{x}) = \mathtt{T}$$

$$\frac{\begin{array}{l}\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{E}_0^b : \mathtt{C} \diamond \\ \forall i \in 1..n \ \Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{E}_i^b : \mathtt{T}_i \diamond \\ \forall i \in 1..n \ \Gamma^{\mathrm{s}} \vdash \mathtt{T}_i \leq \mathtt{T}_i' \\ \Gamma^{\mathrm{s}} \vdash \mathtt{C} \lhd \mathtt{C}' \ \mathtt{T}' \ m(\mathtt{T}_1' \ldots \mathtt{T}_n')\end{array}}{\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{E}_0^b.m \ll \boxed{\mathtt{C}'}.\mathtt{T}' \ (\mathtt{T}_1' \ldots \mathtt{T}_n') \gg (\mathtt{E}_1^b, \ldots, \mathtt{E}_n^b) : \mathtt{T}' \diamond}$$

Figure 2.8: Binary type-judgment

$$\frac{}{\Gamma^{s} \vdash \texttt{int} \leq \texttt{int}}$$

$$\frac{}{\Gamma^{s} \vdash \texttt{C} \leq \texttt{C}}\ \texttt{C} \in \mathit{Def}(\Gamma^{s})$$

$$\frac{\Gamma^{s} \vdash \texttt{C}' :: \_}{\Gamma^{s} \vdash \texttt{C} \leq \texttt{C}'}\ \Gamma^{s}(\texttt{C}) = \langle \texttt{C}', \_ \rangle$$

$$\frac{\Gamma^{s} \vdash \texttt{C} \leq \texttt{C}' \quad \Gamma^{s} \vdash \texttt{C}' \leq \texttt{C}''}{\Gamma^{s} \vdash \texttt{C} \leq \texttt{C}''}$$

$$\frac{\forall i \in 1..n\ \Gamma^{s} \vdash \texttt{T}_i \leq \texttt{T}'_i}{\Gamma^{s} \vdash \texttt{T}_1 \ldots \texttt{T}_n \leq \texttt{T}'_1 \ldots \texttt{T}'_n}$$

$$\frac{\Gamma^{s} \vdash \texttt{C} \leq \texttt{C}_k}{\Gamma^{s} \vdash \{\texttt{C}_1\ \texttt{NH}_1, \ldots, \texttt{C}_n\ \texttt{NH}_n\} \lhd \texttt{C}\ \texttt{NH}}\ k \in 1..n \wedge \texttt{NH} = \texttt{NH}_k$$

$$\frac{\Gamma^{s} \vdash \texttt{C} :: \texttt{ANHS} \quad \Gamma^{s} \vdash \texttt{ANHS} \lhd \texttt{ANH}}{\Gamma^{s} \vdash \texttt{C} \lhd \texttt{ANH}}$$

Figure 2.9: Implementation and widening

Indeed, as already mentioned, in a binary method invocation the descriptor annotation indicates exactly which method to look for, and we only have to check that the types of the receiver expression and of the parameters are subtypes of those specified in the descriptor, and the class of the actual receiver $\texttt{C}$ still implements such a method (premise $\Gamma^{s} \vdash \texttt{C} \lhd \texttt{C}'\ \texttt{T}'\ m(\texttt{T}'_1 \ldots \texttt{T}'_n)$, see Figure 2.9 for the definition of this judgment); this informally means that class $\texttt{C}$ must inherit method $\texttt{T}'\ m(\texttt{T}'_1 \ldots \texttt{T}'_n)$ from $\texttt{C}'$ or any superclasses of $\texttt{C}'$ (of course, if $\texttt{C}' = \texttt{C}$, then the method can also be defined in $\texttt{C}$ itself). Note that this corresponds to the run-time check performed by the JVM when invoking methods, therefore requiring method $\texttt{T}'\ m(\texttt{T}'_1 \ldots \texttt{T}'_n)$ to be exactly defined in $\texttt{C}'$ would be too strong.

The judgment $\Gamma^{s} \vdash \texttt{int} \leq \texttt{int}$ is trivially valid in every Γ^{s}.

Every class $\texttt{C}$ defined in Γ^{s} is a subtype of itself (second rule) and of its (direct) superclass (third rule). Note that every class in Γ^{s} is considered subclass of itself, even if its inheritance hierarchy is cyclical, because this does not lead to wrong type assumption. Vice versa, a class $\texttt{C}$ is considered subclass of $\texttt{C}'$ only if its inheritance relation is acyclic.

Subtyping relation is transitive, fourth rule. The fifth rule extends the subtype relation to tuples of types.

$$b_1 \wedge^E b_2 =_{\text{def}} \begin{cases} b_1 & \text{if } b_1 \in \varepsilon \\ b_2 & \text{if } b_2 \in \varepsilon, b_1 \notin \varepsilon \\ Ok & \text{otherwise} \end{cases}$$

Figure 2.10: Exception-aware disjunction

$$\begin{array}{l}
\mathit{MBody}(ce_b, \mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}) = \\
\begin{cases}
\mathit{NoSuchMethod} \text{ if } \mathtt{C} = \mathtt{Object} \\
\mathit{ClassNotFound} \text{ if } \mathtt{C} \neq \mathtt{Object}, \mathtt{C} \notin \mathit{Def}(ce_b) \\
\langle \mathtt{E}^b, \mathtt{x}_1 \ldots \mathtt{x}_n \rangle \text{ if } \mathtt{T}\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\{\mathtt{return}\ \mathtt{E}^b;\} \in \mathit{code}(ce_b(\mathtt{C})) \\
\mathit{MBody}(ce_b, \mathit{superclass}(ce_b(\mathtt{C})), \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n, \mathtt{T}) \text{ otherwise}
\end{cases} \\
\\
\mathit{code}(\langle _, _, \mathtt{MDS}^b, _ \rangle) = \mathtt{MDS}^b \\
\\
\mathit{superclass}(\langle _, \mathtt{C}', _, _ \rangle) = \mathtt{C}'
\end{array}$$

Figure 2.11: Definition of *MBody*

The judgment $\Gamma^{\mathrm{s}} \vdash \{\mathtt{C}_1\ \mathtt{NH}_1, \ldots, \mathtt{C}_n\ \mathtt{NH}_n\} \lhd \mathtt{C}\ \mathtt{NH}$ is valid whenever one of the annotated method headers $\mathtt{C}_i\ \mathtt{NH}_i$ "implements" the annotated method header $\mathtt{C}\ \mathtt{NH}$, that is, there is a method with the same header in $\mathtt{C}$ or any of its superclasses. If a class has full type $\mathtt{ANHS}$ and the latter implements the method header $\mathtt{ANH}'$, then the class is said to implement $\mathtt{ANH}'$.

The last rule defines the judgment $\Gamma^{\mathrm{s}} \vdash \mathtt{C} \lhd \mathtt{ANH}'$, that is valid whenever class $\mathtt{C}$ implements $\mathtt{ANH}'$.

We define now execution and verification of binary fragments. We anticipate some auxiliary definitions.

Figure 2.10 shows the definition of the operation $\wedge^E$, whose arguments are either set of class names or exceptions. This operation is similar to a boolean conjunction; it returns *Ok* when both arguments are *Ok* and one of its argument when it is an exception (giving priority to the left argument).

The function $\mathit{MBody}(ce_b,\mathtt{C},\mathtt{m},\mathtt{T}_1 \ldots \mathtt{T}_n,\mathtt{T})$, defined in Figure 2.11, models method look-up at run-time; it searches in the binary environment ce_b for the body of a method named $\mathtt{m}$ whose argument types are $\mathtt{T}_1 \ldots \mathtt{T}_n$ and return type is $\mathtt{T}$, starting from class $\mathtt{C}$. If such a method is not found in $\mathtt{C}$, then it is searched, recursively, in its superclass.

The result of *MBody* can be either the body of the method (if found) or an exception if

$$
\begin{array}{lcl}
[\cdot]^{Exp} & ::= & [\cdot].m \ll \boxed{\mathtt{C}}.\mathtt{T}\ (\mathtt{T}_1 \ldots \mathtt{T}_n) \gg (\mathtt{E}^b{}_1, \ldots, \mathtt{E}^b{}_n) \mid \\
 & & \mathtt{new\ C().m} \ll \boxed{\mathtt{C}'}.\mathtt{T}\ (\mathtt{T}_1 \ldots \mathtt{T}_n) \gg (v_1, \ldots, v_{i-1}, [\cdot], \mathtt{E}^b_{i+1}, \ldots, \mathtt{E}^b_n) \\
[\cdot]^{Type} & ::= & \mathtt{new\ C().m} \ll [\cdot].\mathtt{T}\ (\mathtt{T}_1 \ldots \mathtt{T}_n) \gg (v_1, \ldots, v_n) \mid \\
 & & \mathtt{new}\ [\cdot]()
\end{array}
$$

Figure 2.12: Contexts

$$
\mathit{WeakSubtype}(\mathtt{T}, \mathtt{T}', ce_b) = \begin{cases}
\mathit{Ok} & \text{if } \mathtt{T} = \mathtt{T}' \\
\varepsilon & \text{if } \mathtt{T} \neq \mathtt{T}', \mathit{Supertypes}(\mathtt{T}, ce_b) = \varepsilon \\
\mathit{Ok} & \text{if } \mathtt{T} \neq \mathtt{T}', \mathtt{T}' \in \mathit{Def}(ce_b), \\
 & \quad \mathtt{T}' \in \mathit{Supertypes}(\mathtt{T}, ce_b) \\
\mathit{VerifyError} & \text{if } \mathtt{T} \neq \mathtt{T}', \mathtt{T}' \in \mathit{Def}(ce_b), \\
 & \quad \mathtt{T}' \notin \mathit{Supertypes}(\mathtt{T}, ce_b) \\
\mathit{ClassNotFound} & \text{otherwise}
\end{cases}
$$

$$
\begin{array}{l}
\mathit{Supertypes}(\mathtt{T}, ce_b) = \mathit{Supertypes}_{aux}(\mathtt{T}, \emptyset, ce_b) \\
\mathit{Supertypes}_{aux}(\mathtt{T}, \mathit{LC}, ce_b) = \\
\begin{cases}
\mathit{ClassCircularityError} \text{ if } \mathtt{T} \in \mathit{LC} \\
\mathit{ClassNotFound} \text{ if } \mathtt{T} \notin \mathit{Def}(ce_b) \cup \{\mathtt{int}, \mathtt{Object}\} \\
\{\mathtt{Object}\} \cup \mathit{LC} \text{ if } \mathtt{T} = \mathtt{Object} \\
\{\mathtt{int}\} \text{ if } \mathtt{T} = \mathtt{int} \\
\mathit{Supertypes}_{aux}(\mathit{superclass}(ce_b(\mathtt{T})), \{\mathtt{T}\} \cup \mathit{LC}, ce_b) \text{ if } \mathtt{T} \notin \mathit{LC}, \mathtt{T} \in \mathit{Def}(ce_b)
\end{cases}
\end{array}
$$

Figure 2.13: *WeakSubtype* and *Supertypes* definitions

it is not found. There are two kinds of error that can happen during the method look-up: if $\mathtt{C} = \mathtt{Object}$ then this means that the method cannot be found[15], so the exception *NoSuchMethod* is returned. Otherwise, if C cannot be found in the binary environment, then the exception *ClassNotFound* is returned.

In Figure 2.12 we introduce two kinds of contexts: expression contexts $[\cdot]^{Exp}$ and type contexts $[\cdot]^{Type}$. In rewrite semantics, given in Figure 2.15, the former are used to propagate execution to sub-expressions and the latter to verify class references in order to unbox them (making it possible the continuation of the execution). The function *WeakSubtype* is used, at verification time, to check whether a type is subtype of another; $\mathit{WeakSubtype}(\mathtt{T}, \mathtt{T}', ce_b)$, defined in Figure 2.13, returns *Ok* when the type T is a subtype of $\mathtt{T}'$ in the binary environment ce_b or an appropriate exception when it is not. The subtype is "weak" because of the special case $\mathtt{T} = \mathtt{T}'$: any type T is *always* considered subtype of itself; it does not even

[15]For simplicity, we ignore all the predefined methods of Object, defined in 4.3.2 of [GJSB00].

matter whether $\mathtt{T}$ exists or not in the binary environment ce_b. In all the other cases both $\mathtt{T}$ and $\mathtt{T}'$ must exist, otherwise an exception *ClassNotFound* is returned. When both exist in ce_b, the auxiliary function *Supertypes* is used to check the relationship between $\mathtt{T}$ and $\mathtt{T}'$; indeed, $\mathtt{T}$ is subtype of $\mathtt{T}'$ iff $\mathtt{T}'$ is a supertype of $\mathtt{T}$. The function $\mathit{Supertypes}(\mathtt{T}, ce_b)$ can either return the set of supertypes of $\mathtt{T}$ in ce_b, when they can be computed, or an exception in case of error. There are two possible error situations: when a class (directly or indirectly) extends itself and when a parent class is not found in ce_b; in these cases the exceptions *ClassCircularityError* and *ClassNotFound* are, respectively, returned. Figure 2.14 shows the verification judgments. The top-level judgment $\vdash_{ce_b} \mathtt{C} : \mathit{Ok}$ is valid whenever the class $\mathtt{C}$ can be verified in the binary environment ce_b, otherwise it is valid a judgment $\vdash_{ce_b} \mathtt{C} : \varepsilon$, where ε indicates the error occurred in the verification steps. Indeed, it can be proved that the verification process is deterministic and always terminates (either with *Ok* or with an exception).

Note the interesting relation between verification and typechecking of binaries as defined in Figure 2.8; the former corresponds to dynamic typechecking of binaries, whereas the latter to static typechecking, and, hence, is more conservative than the former. This relation is formalized by Theorem 2.3.4 below.

When a class exists in the current binary environment ce_b its verification consists of: checking its superclass, checking that there are no different method declarations with the same signature (that is, name and parameter types) in the code of the class (predicate *noDup*) and verifying that all method declarations ($\vdash_{ce_b} \mathtt{MD}^b_i : b_i$) and the main expression ($\emptyset \vdash_{ce_b} \mathtt{E}^b : \langle \_, b \rangle$) are *Ok*.

When a class does not exists in a binary environment ce_b its verification simply gives *ClassNotFound* (second metarule).

The judgment $\Pi \vdash_{ce_b} \mathtt{E}^b : \langle \mathtt{T}, b \rangle$ is valid whenever the expression $\mathtt{E}^b$ in a binary context ce_b and local type environment Π has type $\mathtt{T}$ and the result of its verification is b. The value b can be either *Ok*, when the verification succeeds, or an exception, indicating the problem, when the verification fails. In this latter case the value of $\mathtt{T}$ is immaterial. A local type environment Π is a (partial) function from argument names to types.

The verification of a method declaration (third rule) succeeds when the verification of its body succeeds and the type of the body is a subtype of the declared return type.

The verification of a method invocation succeeds when the number of arguments coincides with the number of parameter types in the method descriptor, the verification of the receiver and of each argument type succeeds and the type of the receiver and of each argument is a (weak) subtype of the corresponding type contained in the method descriptor (fourth rule).

The fifth rule covers the case when the numbers of arguments differs (side condition $k \neq n$).

$$\frac{\begin{array}{l}\vdash_{ce_b} \mathtt{C}' : b_0 \\ \forall i \in 1..n \;\; \vdash_{ce_b} \mathtt{MD}^b_i : b_i \\ \emptyset \vdash_{ce_b} \mathtt{E}^b : \langle \_, b\rangle\end{array}}{\vdash_{ce_b} \mathtt{C} : \mathit{noDup}(\mathtt{MD}^b{}_1 \ldots \mathtt{MD}^b{}_n) \bigwedge^E_{i\in 0..n} b_i \wedge^E b} \; ce_b(\mathtt{C}) = \langle \mathtt{C}, \mathtt{C}', \mathtt{MD}^b{}_1 \ldots \mathtt{MD}^b{}_n, \mathtt{E}^b\rangle$$

$$\frac{}{\vdash_{ce_b} \mathtt{C} : \mathit{ClassNotFound}} \; \mathtt{C} \notin \mathit{Def}(ce_b)$$

$$\frac{\{\mathtt{x}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash_{ce_b} \mathtt{E}^b : \langle \mathtt{T}, b\rangle}{\vdash_{ce_b} \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \{\mathtt{return}\ \mathtt{E}^b;\} : b \wedge^E \mathit{WeakSubtype}(\mathtt{T}, \mathtt{T}_0, ce_b)}$$

$$\frac{\forall i \in 0..n \; \Pi \vdash_{ce_b} \mathtt{E}^b_i : \langle \mathtt{T}_i, b_i\rangle}{\Pi \vdash_{ce_b} \mathtt{E}^b_0.\mathtt{m} \ll \boxed{\mathtt{T}'_0}.\mathtt{T}\ (\bar{\mathtt{T}}') \gg (\mathtt{E}^b{}_1, \ldots, \mathtt{E}^b{}_n) : \langle \mathtt{T}, \bigwedge^E_{i\in 0..n} s_i\rangle} \; \begin{array}{l}\bar{\mathtt{T}}' = \mathtt{T}'{}_1 \ldots \mathtt{T}'{}_n \\ s_i = b_i \wedge^E \\ \quad \mathit{WeakSubtype}(\mathtt{T}_i, \mathtt{T}'_i, ce_b)\end{array}$$

$$\frac{}{\Pi \vdash_{ce_b} \mathtt{E}^b_0.\mathtt{m} \ll \boxed{\mathtt{T}'_0}.\mathtt{T}\ (\mathtt{T}'{}_1 \ldots \mathtt{T}'{}_n) \gg (\mathtt{E}^b{}_1, \ldots, \mathtt{E}^b{}_k) : \langle \mathtt{T}, \mathit{VerifyError}\rangle} \; k \neq n$$

$$\frac{}{\Pi \vdash_{ce_b} n : \langle \mathtt{int}, \mathit{Ok}\rangle} \; n = 0, 1, -1, 2, -2, \ldots$$

$$\frac{}{\Pi \vdash_{ce_b} \mathtt{new}\ \boxed{\mathtt{C}}\,() : \langle \mathtt{C}, \mathit{Ok}\rangle}$$

$$\frac{}{\Pi \vdash_{ce_b} \mathtt{x} : \langle \Pi(\mathtt{x}), \mathit{Ok}\rangle} \; \mathtt{x} \in \mathit{Def}(\Pi)$$

$$\mathit{noDup}(\mathtt{MD}^b{}_1 \ldots \mathtt{MD}^b{}_n) = \begin{cases} \mathit{Ok} & \text{if } \forall i, j \in 1..n \; \mathit{methSig}(\mathtt{MD}^b_i) = \mathit{methSig}(\mathtt{MD}^b_j) \implies i = j \\ \mathit{VerifyError} & \text{otherwise}\end{cases}$$

$$\mathit{methSig}(\mathtt{T}\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\{\mathtt{E}^b\}) = \mathtt{T}\ \mathtt{m}(\mathtt{T}_1 \ldots \mathtt{T}_n)$$

Figure 2.14: Verification

$$\frac{\vdash_{ce_b} \mathtt{C} : Ok \quad \mathtt{E}^b \overset{*}{\leadsto}_{ce_b} V}{\mathtt{C} \leadsto_{ce_b} V} \; ce_b(\mathtt{C}) = \langle \_, \_, \_, \mathtt{E}^b \rangle$$

$$\frac{\vdash_{ce_b} \mathtt{C} : \varepsilon}{\mathtt{C} \leadsto_{ce_b} \varepsilon}$$

$$\frac{\vdash_{ce_b} \mathtt{C} : \varepsilon}{[\boxed{\mathtt{C}}]^{Type} \leadsto_{ce_b} \varepsilon}$$

$$\frac{\vdash_{ce_b} \mathtt{C} : Ok}{[\boxed{\mathtt{C}}]^{Type} \leadsto_{ce_b} [\mathtt{C}]^{Type}}$$

$$\frac{\mathtt{E}^b \leadsto_{ce_b} \mathtt{E}_1^b}{[\mathtt{E}^b]^{Exp} \leadsto_{ce_b} [\mathtt{E}_1^b]^{Exp}} \; \mathtt{E}_1^b \neq \varepsilon$$

$$\frac{\mathtt{E}^b \leadsto_{ce_b} \varepsilon}{[\mathtt{E}^b]^{Exp} \leadsto_{ce_b} \varepsilon}$$

$$\frac{}{\mathtt{new}\ \mathtt{C}().m \ll \mathtt{C}'.\mathtt{T}\ (\bar{\mathtt{T}}) \gg (v_1, \ldots, v_n) \leadsto_{ce_b} \varepsilon} \; \begin{array}{l} \bar{\mathtt{T}} = \mathtt{T}_1 \ldots \mathtt{T}_n \\ MBody(ce_b, \mathtt{C}', \mathtt{m}, \bar{\mathtt{T}}, \mathtt{T}) = \varepsilon \end{array}$$

$$\frac{\mathtt{E}^b[v_1/\mathtt{x}_1, \ldots, v_n/\mathtt{x}_n] \leadsto_{ce_b} \mathtt{E}_1^b}{\mathtt{new}\ \mathtt{C}().m \ll \mathtt{C}'.\mathtt{T}\ (\bar{\mathtt{T}}) \gg (v_1, \ldots, v_n) \leadsto_{ce_b} \mathtt{E}_1^b} \; \begin{array}{l} \bar{\mathtt{T}} = \mathtt{T}_1 \ldots \mathtt{T}_n \\ MBody(ce_b, \mathtt{C}', \mathtt{m}, \bar{\mathtt{T}}, \mathtt{T}) \neq \varepsilon \\ MBody(ce_b, \mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}, \mathtt{T}) = \langle \mathtt{E}^b, \mathtt{x}_1 \ldots \mathtt{x}_n \rangle \end{array}$$

Figure 2.15: Rewriting

Note that if a binary fragment is the result of the compilation of a source fragment, the number of arguments is indeed equal to the number of parameter types in the descriptor; such a mismatch may only be found in "malicious" binary fragments.

Figure 2.15 shows the rewriting rules for the program execution. The first two rules deal with the execution of the main method of a class `C`; the former covers the case when class `C` is verified, whereas the latter considers the case when `C` does not pass verification.

The third and fourth rules cover the loading/verification process. The former is used in case of error: the whole term is rewritten in the exception thrown by the verifier. The latter is used when the verification is carried out successfully; in this case the term is rewritten in itself except for the reference to the class `C` that is unboxed.

The third rule is just the standard closure.

The fourth rule propagates an exception rewriting an entire term containing an exception ε in the exception itself.

The fifth and the sixth rules deal with method invocation. When the method cannot be found starting the search from the class contained in the method descriptor the entire expression is rewritten in the exception; otherwise a second invocation to *MBody*, passing as starting class the dynamic type of the receiver, returns the method body and the name of the arguments. These information are used to expand the method invocation.

2.3.1 Main results

We prove three main theorems claiming the safety of source and binary typechecking and of the safe compilation schema, respectively; the former two theorems are necessary for proving the latter.

Theorem 2.3.1 (Safe Source Typechecking) *For all type environments Γ^s, sources S and binaries B, if $\Gamma^s \vdash S \leadsto B$, then $\Gamma^s \vdash B \diamond$.*

Theorem 2.3.2 (Safe Binary Typechecking) *Let $\langle ce_b, ce_s \rangle$ and C be a compilation environment and a class name, respectively. For all values V, if $\forall C_1 \in \mathcal{D}(\langle ce_b, ce_s \rangle, \{C\})\ \ \mathcal{T}(\langle ce_b, ce_s \rangle) \vdash ce_b(C_1) \diamond$ and $C \leadsto_{ce_b} V$, then V is not an exception.*

We can state now the main property of the safe compilation schema: if a set of classes is successfully compiled w.r.t. the safe schema, then the execution of any binary produced by such compilation in the updated binary environment never throws a linkage exception.

Theorem 2.3.3 (Safety) *Let $\langle ce_b, ce_s \rangle$, CS and ce'_b be a compilation environment, a set of class names and a binary environment, respectively. For all class names $C \in Def(ce'_b)$ and values V, if $\mathcal{C}(\langle ce_b, ce_s \rangle, CS) = ce'_b$ and $C \leadsto_{ce_b[ce'_b]} V$, then V is not an exception.*

2.3.2 Proofs (sketched)

Safe Source Typechecking: The only way to deduce $\Gamma^s \vdash \mathtt{S} \leadsto \mathtt{B}$ is applying the first rule in Figure 2.5, so whenever $\Gamma^s \vdash \mathtt{S} \leadsto \mathtt{B}$ is deducible, $\Gamma^s \vdash \mathtt{C} ::_\diamond \_$ is deducible too, and $\Gamma^s \vdash \mathtt{B} \diamond$ can be deduced if:

- $\Gamma^s \vdash \mathtt{MDS}^s \leadsto \mathtt{MDS}^b \implies \Gamma^s \vdash \mathtt{MDS}^b \diamond$ and,

- $\Gamma^{\mathrm{s}}; \emptyset \vdash \mathtt{E}^s : \_ \rightsquigarrow \mathtt{E}^b \implies \Gamma^{\mathrm{s}}; \emptyset \vdash \mathtt{E}^b : \_ \diamond$.

Both implications rely on the implication:

$$\Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{E}^s : \mathtt{T} \rightsquigarrow \mathtt{E}^b \Rightarrow \Gamma^{\mathrm{s}}; \Pi \vdash \mathtt{E}^b : \mathtt{T} \diamond$$

which trivially holds for all kinds of expression except for method invocations. The method invocation case relies on the following:

$$<\bar{\mathtt{T}}', \mathtt{T}'> = \mathit{methRes}(\Gamma^{\mathrm{s}}, \mathtt{C}, m, \bar{\mathtt{T}}) \implies \begin{cases} \Gamma^{\mathrm{s}} \vdash \bar{\mathtt{T}} \leq \bar{\mathtt{T}}' \\ \Gamma^{\mathrm{s}} \vdash \mathtt{C} \lhd \mathtt{C}\ \mathtt{T}'\ m(\bar{\mathtt{T}}') \end{cases}$$

The first part of the implication trivially holds by definition of *appMeth* (directly used by *methRes*), see Figure 2.6. The latter follows by two facts:

- *methRes* selects the most specific method among the ones in ANHS if $\Gamma^{\mathrm{s}} \vdash \mathtt{C} :: \mathtt{ANHS}$,
- for any annotation $\mathtt{C}'$ in ANHS it must be $\Gamma^{\mathrm{s}} \vdash \mathtt{C}' \leq \mathtt{C}$.

Safe Binary Typechecking: Safety of binary typechecking comes from the following two theorems, the former connecting static with dynamic binary typechecking (that is, the binary typechecking judgment $\Gamma^{\mathrm{s}} \vdash \mathtt{B} \diamond$ with the verification judgment $\vdash_{ce_b} \mathtt{C} : b$), the latter expressing subject reduction for binary expressions. The former can be proved by induction over the rules for binary typechecking, while the latter can be proved by induction over the rewriting rules for binary expressions.

Theorem 2.3.4 (Binary Typechecking Implies Verification) *Let $\langle ce_b, ce_s\rangle$ and C be a compilation environment and a class name, respectively, s.t. the following condition holds: $\mathcal{D}(\langle ce_b, ce_s\rangle, \{C\}) \subseteq \mathit{Def}(ce_b)$. If $\mathcal{T}(\langle ce_b, ce_s\rangle) \vdash ce_b(C) \diamond$, then $\vdash_{ce_b} C$: Ok.*

Note that the converse implication does not hold, since typechecking at (dynamic) load time is more accurate than that at compile time. For instance, typechecking of a binary declaration of a class C requires the check of all classes explicitly mentioned in C, whereas the JVM only checks those classes that are actually needed by that particular execution.

Theorem 2.3.5 (Binary Subject Reduction) *Let $\langle ce_b, ce_s\rangle$ and $\mathtt{E}^b$ be a compilation environment and a binary expression, respectively. If $\mathcal{T}(\langle ce_b, ce_s\rangle); \emptyset \vdash \mathtt{E}^b : T\diamond$ and $\mathtt{E}^b \rightsquigarrow_{ce_b} \mathtt{E}^b_1$, then there exists a type T_1 s.t. $\mathcal{T}(\langle ce_b, ce_s\rangle); \emptyset \vdash \mathtt{E}^b_1 : T_1 \diamond$ and $\mathcal{T}(\langle ce_b, ce_s\rangle) \vdash T_1 \leq T$.*

We are now able to prove safety of binary typechecking. Let us assume that $\forall \mathtt{C}' \in \mathcal{D}(\langle ce_b, ce_s\rangle, \{\mathtt{C}\})\ \ \mathcal{T}(\langle ce_b, ce_s\rangle) \vdash ce_b(\mathtt{C}')\diamond$ and $\mathtt{C} \rightsquigarrow_{ce_b} V$. From the first assumption we easily deduce $\mathcal{D}(\langle ce_b, ce_s\rangle, \{\mathtt{C}\}) \subseteq \mathit{Def}(ce_b)$ and $\mathcal{T}(\langle ce_b, ce_s\rangle) \vdash ce_b(\mathtt{C})\diamond$ (since, trivially, $\mathtt{C} \in \mathcal{D}(\langle ce_b, ce_s\rangle, \{\mathtt{C}\})$).
As a consequence, Theorem 2.3.4 can be applied, therefore $\vdash_{ce_b} \mathtt{C} : \mathit{Ok}$ holds. This means that $\mathtt{C} \rightsquigarrow_{ce_b} V$ has been deduced by instantiating the first (and not the second) meta-rule in Figure 2.15, so $\mathtt{E}^b \overset{*}{\rightsquigarrow}_{ce_b} V$ must hold. Furthermore, the validity of $\mathcal{T}(\langle ce_b, ce_s\rangle) \vdash ce_b(\mathtt{C})\diamond$ implies the validity of $\mathcal{T}(\langle ce_b, ce_s\rangle); \emptyset \vdash \mathtt{E}^b : \mathtt{T}\diamond$, since there is only one meta-rule that can be instantiated in Figure 2.8. Therefore we can apply Theorem 2.3.5 and deduce the validity of $\mathcal{T}(\langle ce_b, ce_s\rangle); \emptyset \vdash V : \mathtt{T}'\diamond$, with $\mathtt{T}'$ subtype of $\mathtt{T}$. Since exceptions do not typecheck (see rules in Figure 2.8), we can conclude that V is not an exception.

Safety: To prove safety we need two lemmas claiming that both the dependency and type extraction functions are invariant w.r.t. source typechecking. These lemmas can be proved by induction over the definition of $\mathcal{D}$ and $\mathcal{T}$, respectively.

In what follows, let $ce_{s\setminus \mathrm{C}}$ denotes the partial function obtained by restricting the definition domain of ce_s to the set $\mathit{Def}(ce_s) \setminus \{\mathtt{C}\}$.

Lemma 2.3.6 *Let $\langle ce_b, ce_s\rangle$, $\mathtt{C}$ and $\mathtt{B}$ be a compilation environment, a class name, and a binary fragment, respectively. If $\mathcal{T}(\langle ce_b, ce_s\rangle) \vdash ce_s(\mathtt{C}) \rightsquigarrow \mathtt{B}$, then for all class name $\mathtt{C}_1$ the following equality holds:*

$$\mathcal{D}(\langle ce_b, ce_s\rangle, \{\mathtt{C}_1\}) = \mathcal{D}(\langle ce_b[\mathtt{C} \mapsto \mathtt{B}], ce_{s\setminus \mathtt{C}}\rangle, \{\mathtt{C}_1\}).$$

Lemma 2.3.7 *Let $\langle ce_b, ce_s\rangle$, $\mathtt{C}$ and $\mathtt{B}$ be a compilation environment, a class name, and a binary fragment, respectively. If $\mathcal{T}(\langle ce_b, ce_s\rangle) \vdash ce_s(\mathtt{C}) \rightsquigarrow \mathtt{B}$, then the following equality holds:*

$$\mathcal{T}(\langle ce_b, ce_s\rangle) = \mathcal{T}(\langle ce_b[\mathtt{C} \mapsto \mathtt{B}], ce_{s\setminus \mathtt{C}}\rangle).$$

Now assume that $\mathcal{C}(\langle ce_b, ce_s\rangle, \mathtt{CS}) = ce'_b$. By virtue of the top-level rule in Figure 2.1, the following judgments are valid:

$$\forall \mathtt{C} \in \mathtt{CS}_b\ \Gamma^{\mathrm{s}} \vdash ce_b(\mathtt{C})\diamond$$
$$\forall \mathtt{C} \in \mathtt{CS}_s\ \Gamma^{\mathrm{s}} \vdash ce_s(\mathtt{C}) \rightsquigarrow \mathtt{B}_{\mathtt{C}}$$

where $\mathtt{CS}_b = \mathtt{CS}_d \cap \mathit{Def}(ce_b)$, $\mathtt{CS}_s = \mathtt{CS}_d \cap \mathit{Def}(ce_s)$, $\mathtt{CS}_d = \mathcal{D}(\langle ce_b, ce_s\rangle, \mathtt{CS})$ and $\Gamma^{\mathrm{s}} = \mathcal{T}(\langle ce_b, ce_s\rangle)$. Furthermore, $ce'_b(\mathtt{C}) = \{\mathtt{C} \mapsto \mathtt{B}_{\mathtt{C}} \mid \mathtt{C} \in \mathtt{CS}_s\}$.

By Theorem 2.3.1, $\Gamma^{\mathrm{s}} \vdash ce'_b(\mathtt{C})\diamond$ for all $\mathtt{C} \in \mathtt{CS}_s$, therefore we can easily deduce $\Gamma^{\mathrm{s}} \vdash ce_b[ce'_b](\mathtt{C})\diamond$ for all $\mathtt{C} \in \mathtt{CS}_d$.

Let us now prove the main theorem by assuming that $\mathtt{C}$ is a class name in $\mathtt{CS}_b$ (recall that $\mathit{Def}(ce'_b) = \mathtt{CS}_b$) and that $\mathtt{C} \rightsquigarrow_{ce_b[ce'_b]} V$ for a certain value V. By lemmas 2.3.6 and 2.3.7

and by induction on the cardinality of $\texttt{CS}$, $\mathcal{D}(\langle ce_b, ce_s\rangle, \{\texttt{C}\}) = \mathcal{D}(\langle ce_b[ce'_b], ce_{s\setminus\texttt{CS}}\rangle, \{\texttt{C}\})$ and $\mathcal{T}(\langle ce_b, ce_s\rangle) = \mathcal{T}(\langle ce_b[ce'_b], ce_{s\setminus\texttt{CS}}\rangle)$. Therefore we can apply Theorem 2.3.2 and conclude that V cannot be an exception.

Chapter 3

True separate compilation and selective recompilation

In Chapter 2 (and in [ALZ02]) we have proposed a formalization of the Java compilation process where the judgment corresponding to intra-checking is clearly isolated from other components (extraction of the type environment from the program and determination of the fragments on which compilation is propagated). However, each class is still intra-checked against a unique standard type environment extracted from the compilation environment. With this approach, inter-checking trivially succeeds, since the type environment is directly extracted from the code and is the same for all fragments. Anyway, a fragment can be successfully intra-checked using "less" information. In this chapter we introduce a formal system which permits to derive typings which are stronger than those of the standard type systems for Java, by introducing the notions of *nonstandard type assumption* and *entailment of type environments*. The former allows the user to specify fine-grain requirements on the source fragments which need to be compiled in isolation, whereas the latter syntactically captures the concept of stronger type environment.

One of the most important advantages of this approach consists in a better support for selective recompilation; indeed, based on the formal system, it is possible to define an algorithm able to avoid the unnecessary recompilation steps which are usually performed by the Java compilers. This particular application is exploited in Section 3.3.

3.1 An informal presentation

This section is a gentle introduction to the system formally defined in Section 3.2. More precisely, the two basic notions of *nonstandard* type assumption and *entailment* relation between type environments are informally presented and motivated.

The language used in the examples, as in the previous chapter, is a basic subset of Java, where classes can only declare methods. Method overloading is present, but constructor overloading is not and it is assumed that every class has, as the only constructor, the default (parameterless) one.

Nonstandard Type Assumptions

Let us consider a declaration of the class H:

```
class H extends P {
  int g(P p) {
    return p.f(new H()) ;
  }
  int m() {
    return new H().g(new P()) ;
  }
  U id(U u) {
    return u ;
  }
  X em(Y y) {
    return y ;
  }
}
```

and analyze under which assumptions class H can be successfully compiled. If we take the approach of the SDK compiler, then we would need to impose rather strong requirements on all classes used by H, by asking for the most detailed type information about such classes.

In our system this corresponds to compile H in a type environment Γ^{s} which contains standard type assumptions on the classes P, U, X and Y. For instance, if Γ^{s} is defined by:

$$\Gamma^{\mathrm{s}} = \mathtt{P} \mapsto <\mathtt{Object}, \mathtt{int\ f(Object)}>, \mathtt{U} \mapsto <\mathtt{Object}, >, \mathtt{Y} \mapsto <\mathtt{X}, >, \mathtt{X} \mapsto <\mathtt{Object}, >$$

then we are assuming that class P extends Object and declares only int f(Object), classes U and X both extend Object and are empty, and class Y extends X and is empty. An environment like Γ^{s} containing only standard type assumptions is called a *standard type environment*.

Under the assumptions contained in Γ^{s} class H can be successfully compiled to the following binary fragment $\mathtt{B_h}$:

```
class H extends P {
  int g(P p) {
```

```
    return p.f<<P.int (Object)>>(new H()) ;
  }
  int m() {
    return new H().g<<H.int (P)>>(new P()) ;
  }
  U id(U u) {
    return u ;
  }
  X em(Y y) {
    return y ;
  }
}
```

In our system, as the reader may recall from previous chapter, a binary fragment is just like a source fragment except that invocations contain an *annotation* $\ll$ $\mathtt{C.T}$ $(\mathtt{T_1} \ldots \mathtt{T_n})$ $\gg$ giving the class `C` in which the method is to be found (see [LY99] 5.1), the return type `T`, and the parameter types $\mathtt{T}_1 \ldots \mathtt{T}_n$ of the method which has been selected as most specific at compile time. Indeed, from our perspective the most critical difference between source and binary fragments is type annotations in the method invocations, since it makes the problem of separate compilation (that is, separate typechecking plus code generation) substantially different from that of separate typechecking.

Let us now try to relax the strong assumptions in Γ^{S} by seeking an environment Γ^{NS} containing other kinds of type assumptions which still guarantee that `H` compiles to the same binary fragment $\mathtt{B_h}$, but impose fairly weaker requirements on classes `P`, `U`, `X` and `Y`.

A first basic request is that the compilation environment containing `H` must provide a definition for the four classes which `H` depends on. In our system this is expressed by a nonstandard assumption of the form $\exists\,\mathtt{C}$, therefore Γ^{NS} will contain at least the assumptions $\exists\,\mathtt{P}, \exists\,\mathtt{U}, \exists\,\mathtt{X}, \exists\,\mathtt{Y}$.

Let us now focus on each single class used by `H`.
Class `P`: in order to correctly compile class `H` (into $\mathtt{B_h}$) the following additional assumptions on class `P` must be added to Γ^{NS}:

- $\mathtt{P} \not< \mathtt{H}$: `P` cannot be a proper subtype of `H` since inheritance cannot be cyclic.
- `P`☺`int g(P)`: `P` can be correctly extended with method `int g(P)`; indeed, according to Java rules on method overriding, if `P` has a method `g(P)`, then `g` must have the same return type `int` as declared in `H`. Analogous requirements are needed for the other methods declared in `H`.
- $\mathtt{P.f(H)} \xrightarrow{\mathtt{res}}$ <`Object`, `int`>: invocation of method `f` of an object of type `P` with an argument of type `H`, is successfully resolved to a method with a parameter of type

Object and return type int. This assumption ensures that the body of g in H is successfully compiled to the same bytecode of method g in $\mathtt{B_h}$ (in other words, the same symbolic reference to the method is generated). Note that we do not need to know the class where the method is declared, since the bytecode is annotated with the type of the receiver.

Class U: no additional requirements on U are needed, since the static correctness of method id in H only requires the existence of U.
Classes X and Y: in order to correctly compile class H, class Y must be a subtype of class X, otherwise method em in H would not be statically correct. Therefore we need to add the assumption $\mathtt{Y} \leq \mathtt{X}$.

In conclusion, class H can be successfully compiled to $\mathtt{B_h}$ in the environment Γ^{NS} defined by:

$$\begin{aligned}\Gamma^{\mathrm{NS}} = \;& \exists\,\mathtt{P}, \exists\,\mathtt{U}, \exists\,\mathtt{X}, \exists\,\mathtt{Y}, \mathtt{P} \not\leq \mathtt{H}, \mathtt{Y} \leq \mathtt{X}, \mathtt{P}☺\mathtt{int\ g(P)},\\ & \mathtt{P}☺\mathtt{int\ m()}, \mathtt{P}☺\mathtt{U\ id(U)}, \mathtt{P}☺\mathtt{X\ em(Y)}, \mathtt{P.f(H)} \overset{\mathtt{res}}{\to} <\mathtt{Object}, \mathtt{int}>\end{aligned}$$

Furthermore, Γ^{NS} is weaker than Γ^{S}; for instance, class U must extend Object and be empty in Γ^{S}, while in Γ^{NS} it can extend any class and declare any method. The notion of stronger type environment is syntactically captured by an entailment relation on type environments.

Entailment of Type Environments

Referring to the previous example, in our system the intuition that Γ^{NS} is weaker than Γ^{S} is formalized by the following property: for all $\mathtt{S}, \tau, \mathtt{B}$ if $\Gamma^{\mathrm{NS}} \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$ is provable, then $\Gamma^{\mathrm{S}} \vdash \mathtt{S} : \tau \leadsto \mathtt{B}$ is provable as well. However, since the definition above cannot be directly checked in an effective way, the notion of stronger type environment needs to be captured by an *entailment* relation (that is, a computable relation) between type environments.

For instance, in our system $\Gamma^{\mathrm{S}} \vdash \Gamma^{\mathrm{NS}}$ can be proved. Furthermore, the entailment relation is proved to be *sound*, that is, if $\Gamma_1 \vdash \Gamma_2$ can be proved, then Γ_1 is stronger than Γ_2. In the particular example, we can go further, by showing that Γ^{NS} is actually strictly weaker than Γ^{S}.

Let us add in H the new method `int one(){return 1;}`. After this change, the new code for class H still intra-checks in Γ^{S}, whereas intra-checking of the same code in Γ^{NS} fails. To see this, let us consider the following new declaration for class P:

```
class P extends Object{
  int f(Object o) {
    return 1 ;
  }
  P one() {
    return new P() ;
```

```
  }
}
```

The reader can easily verify that each type assumption in Γ^{NS} about `P` is satisfied by the new version of `P` above, however if we put all classes together we obtain a statically incorrect program, since method `one` is redefined in `H` with a different return type. Therefore Γ^{S} is strictly stronger than Γ^{NS}; from this last claim and from the soundness of the entailment we can deduce $\Gamma^{\text{NS}} \not\models \Gamma^{\text{S}}$.

3.2 Formalization

The language we consider is a rather small but significant subset of Java; indeed, it includes one of the most critical features for separate compilation which is Java *static overloading* ([Car97] – see the end of page 1). The main difference, with respect to the language formalized in the previous chapter, is the absence of the main expression and of boxed class names, since we do not model run-time behavior here.

The syntax of the language is defined in Figure 3.1; the type environments are defined in Figure 3.2. The metavariables `C`, `m`, `x` and `N` range over sets of class, method and parameter names, and integer literals, respectively.

A program `P` is a sequence of source fragments; a source fragment `S` is a class declaration consisting of the name of the class, the name of the superclass and a sequence of method declarations $\mathtt{MDS}^s$. A method declaration $\mathtt{MD}^s$ consists of a method header and a method body (an expression). A method header `MH` consists of a (return) type, a method name and a sequence of parameter types and names. There are four kinds of expression: instance creation, parameter name, integer literal, and method invocation. A type can be either a class name or `int`.

As already mentioned, the bytecode of our language differs from the source code only for method invocations which contain a symbolic reference $\ll \mathtt{C.T}(\bar{\mathtt{T}}) \gg$ to the method to be invoked (see Section 3.1).

A standard type environment Γ^{S} is a possibly empty sequence of type assumptions of the form $\mathtt{C} \mapsto <\mathtt{C}', \mathtt{NHS}>$ with the meaning "`C` extends $\mathtt{C}'$ and declares exactly all methods[1] specified by `NHS`" where $\mathtt{NHS} = \mathtt{NH}_1 \ldots \mathtt{NH}_n$, and for all $i \in 1 \ldots n$ $\mathtt{NH}_i$ is a return type followed by a *method signature* (that is, a method name and a sequence of parameter types). We denote the empty sequence by Λ and use the notation $\mathit{Def}(\Gamma^{\text{S}})$ for the set $\{\mathtt{C} \mid \mathtt{C} \mapsto \tau \in \Gamma^{\text{S}}\}$ (where $\gamma \in \Gamma$ means Γ contains the assumption γ).

[1]For simplicity, in our language instance methods are the only members a class can contain.

$$
\begin{array}{rcll}
\mathtt{P} & ::= & \langle \mathtt{S}_1 \ldots \mathtt{S}_n \rangle & \\
\mathtt{S} & ::= & \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^s\ \} & \\
\mathtt{MDS}^s & ::= & \mathtt{MD}^s_1 \ldots\ \mathtt{MD}^s_n & (n \geq 0) \\
\mathtt{MD}^s & ::= & \mathtt{MH}\ \{\ \mathtt{return\ E}^s;\ \} & \\
\mathtt{MH} & ::= & \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n) & (n \geq 0) \\
\mathtt{E}^s & ::= & \mathtt{new\ C}() \mid \mathtt{x} \mid \mathtt{N} \mid \mathtt{E}^s_0.\mathtt{m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) & \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{int} & \\
\bar{\mathtt{T}} & ::= & \mathtt{T}_1 \ldots \mathtt{T}_n & \\
\bar{\mathtt{B}} & ::= & \langle \mathtt{B}_1 \ldots \mathtt{B}_n \rangle & \\
\mathtt{B} & ::= & \mathtt{class\ C\ extends\ C'}\ \{\ \mathtt{MDS}^b\ \} & \\
\mathtt{MDS}^b & ::= & \mathtt{MD}^b_1 \ldots\ \mathtt{MD}^b_n & (n \geq 0) \\
\mathtt{MD}^b & ::= & \mathtt{MH}\ \{\ \mathtt{return\ E}^b;\ \} & \\
\mathtt{E}^b & ::= & \mathtt{new\ C}() \mid \mathtt{x} \mid \mathtt{N} \mid \mathtt{E}^b_0 \ll \mathtt{C.T}(\bar{\mathtt{T}}) \gg \mathtt{m}(\mathtt{E}^b_1, \ldots, \mathtt{E}^b_n) & (n \geq 0) \\
 & & & \\
\mathtt{NH} & ::= & \mathtt{T\ m}(\bar{\mathtt{T}}) & \\
\mathtt{NHS} & ::= & \{\mathtt{NH}_1, \ldots, \mathtt{NH}_n\} &
\end{array}
$$

Implicit assumptions:

- class names in P are distinct;
- method signatures in $\mathtt{MDS}^s$ and $\mathtt{MDS}^b$ are distinct;
- parameter names in MH are distinct.

Figure 3.1: Syntax

A nonstandard type environment Γ^{NS} is a possibly empty sequence of nonstandard type assumptions of the following kinds:

- $\exists\,\mathtt{T}$ with the meaning "T is defined";
- $\mathtt{T} \leq \mathtt{T}'$ with the meaning "T is a subtype of T′";
- $\mathtt{C.m}(\bar{\mathtt{T}}) \stackrel{res}{\to} \mathtt{<}\bar{\mathtt{T}}', \mathtt{T}\mathtt{>}$ with the meaning "the invocation of method m of an object of type C with arguments of type $\bar{\mathtt{T}}$, is successfully resolved to a method (obviously named m) with parameters of type $\bar{\mathtt{T}}'$ and return type T.
- C☺T $\mathtt{m}(\mathtt{T}_1 \ldots \mathtt{T}_n)$ with the meaning "C can be extended by a subclass having method $\mathtt{T\ m}(\mathtt{T}_1 \ldots \mathtt{T}_n)$ without breaking the Java rule on method overriding";
- $\mathtt{C} \not< \mathtt{C}'$ with the meaning "C is not a proper subtype of C′".

$$
\begin{array}{rcl@{\qquad}rcl}
\mathtt{NH} & ::= & \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) & \Gamma^{\mathrm{NS}} & ::= & {\gamma^{\mathrm{NS}}}_1 \ldots {\gamma^{\mathrm{NS}}}_n \\
\mathtt{NHS} & ::= & \{\mathtt{NH}_1, \ldots, \mathtt{NH}_n\} & \gamma^{\mathrm{NS}} & ::= & \exists\,\mathtt{T}\ \mid \\
\mu & ::= & \ll \mathtt{C.T}\ (\bar{\mathtt{T}}) \gg & & & \mathtt{T} \leq \mathtt{T}'\ \mid \\
\mu\mathtt{s} & ::= & \{\mu_1, \ldots, \mu_n\} & & & \mathtt{C.m}(\bar{\mathtt{T}}) \overset{res}{\rightarrow} <\bar{\mathtt{T}}', \mathtt{T}>\ \mid \\
 & & & & & \mathtt{C}☺\mathtt{NH} \mid \\
\Gamma^{\mathrm{S}} & ::= & {\gamma^{\mathrm{S}}}_1 \ldots {\gamma^{\mathrm{S}}}_n & & & \mathtt{C} \not< \mathtt{C}' \mid \\
\gamma^{\mathrm{S}} & ::= & \mathtt{C} \mapsto \tau & \Gamma & ::= & \gamma_1 \ldots \gamma_n \\
\tau & ::= & <\mathtt{C}, \mathtt{NHS}> & \gamma & ::= & \gamma^{\mathrm{S}} \mid \gamma^{\mathrm{NS}} \\
\bar{\tau} & ::= & \langle \tau_1 \ldots \tau_n \rangle & & &
\end{array}
$$

Figure 3.2: Type environments

Finally, type environments Γ used for separate compilation can contain both standard and nonstandard type assumptions; standard type assumptions are needed for dealing with mutual recursion between classes (see rule for separate compilation of programs in Figure 3.3) and for compatibility with the SDK systems.

Typing rules for separate compilation are defined in Figure 3.3.

The top-level rule defines the compilation of a program P, whose type is $\langle \tau_1 \ldots \tau_n \rangle$, into a set of binary fragments $\langle \mathtt{B}_1 \ldots \mathtt{B}_n \rangle$. The provided environment, Γ, is enriched with the type of the fragments to compile, to deal with mutual recursion. The resulting environment Γ' must be well-formed and the compilation of all the fragments $\mathtt{S}_i$ must be derivable in Γ'. The functions *className* and $\mathcal{T}$ (defined in Figure 2.2, page 24) extract from a class declaration the name and the type of the class, respectively.

The rule which defines the compilation of a single fragment for class C checks that all methods can be compiled, that the superclass $\mathtt{C}'$ can be safely extended with the methods declared in C and that there are no cycles involving C and $\mathtt{C}'$ (existence of the superclass is guaranteed by this last check). The function $\mathcal{T}$ extract from a method declaration the return type and the signature of the method.

The rule for compiling a method declaration checks that the body can be compiled and that the return type and the types of the parameters are defined.

The typing rule for method invocation checks that all sub-expressions can be compiled and that the method can be successfully resolved.

The entailment of type environments is given in Figure 3.4.

Rule (*well-def*) defines well-formed type environments, that is, consistent environments, and relies on the definition of well-formed standard environments. A standard type environment is well-formed if the inheritance relation is acyclic, for each class all its ancestor

$$\frac{\begin{array}{l}\vdash \Gamma'\diamond \\ \Gamma' \vdash \mathtt{S}_i : \tau_i \leadsto \mathtt{B}_i\ \forall i \in 1,\dots,n\end{array}}{\Gamma \vdash \mathtt{P} : \langle \tau_1 \dots \tau_n\rangle \leadsto \langle \mathtt{B}_1\dots\mathtt{B}_n\rangle} \quad \begin{array}{l} \mathtt{P} = \langle \mathtt{S}_1\dots\mathtt{S}_n\rangle \\ \Gamma' = \Gamma, \mathtt{C}_1 \mapsto \tau_1,\dots,\mathtt{C}_n\mapsto\tau_n \\ \mathtt{C}_i = \mathit{className}(\mathtt{S}_i), \tau_i = \mathcal{T}(\mathtt{S}_i)\\ \forall i\in 1,\dots,n\end{array}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{MD}_i^s \leadsto \mathtt{MD}_i^b\ \forall\ i \in 1,\dots,n \\ \Gamma \vdash \mathtt{C}'\text{☺}\mathtt{NH}_i\ \forall\ i \in 1,\dots,n \\ \Gamma \vdash \mathtt{C}' \not\leq \mathtt{C}\end{array}}{\begin{array}{ll}\Gamma \vdash & \mathtt{class\ C\ extends\ C'}\ \{\mathtt{MDS}^s\} : \langle \mathtt{C'}, \mathtt{NHS}\rangle \leadsto \\ & \mathtt{class\ C\ extends\ C'}\ \{\mathtt{MDS}^b\}\end{array}} \quad \begin{array}{l}\mathtt{MDS}^s = \mathtt{MD}_1^s \dots \mathtt{MD}_n^s \\ \mathtt{NHS} = \{\mathtt{NH}_1, \dots \mathtt{NH}_n\} \\ \mathcal{T}(\mathtt{MD}_i^s) = \mathtt{NH}_i \\ \forall\ i \in 1,\dots,n\end{array}$$

$$\frac{\begin{array}{l}\Gamma; \{\mathtt{x}_1 \mapsto \mathtt{T}_1,\dots,\mathtt{x}_n \mapsto \mathtt{T}_n\} \vdash \mathtt{E}^s : \mathtt{T} \leadsto \mathtt{E}^b \\ \Gamma \vdash \mathtt{T} \leq \mathtt{T}_0 \\ \Gamma \vdash \exists\, \mathtt{T}_i\ \forall\ i \in 0,\dots,n\end{array}}{\begin{array}{ll}\Gamma \vdash & \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1,\dots,\mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E}^s;\ \} \leadsto \\ & \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1,\dots,\mathtt{T}_n\ \mathtt{x}_n)\ \{\ \mathtt{return}\ \mathtt{E}^b;\ \}\end{array}} \qquad \frac{}{\Gamma; \Pi \vdash \mathtt{N} : \mathtt{int} \leadsto \mathtt{N}}$$

$$\frac{}{\Gamma; \Pi \vdash \mathtt{x} : \mathtt{T} \leadsto \mathtt{x}}\ \Pi(\mathtt{x}) = \mathtt{T} \qquad \frac{\Gamma \vdash \exists\, \mathtt{C}}{\Gamma; \Pi \vdash \mathtt{new\ C} : \mathtt{C} \leadsto \mathtt{new\ C}}$$

$$\frac{\begin{array}{l}\Gamma; \Pi \vdash \mathtt{E}_0^s : \mathtt{C} \leadsto \mathtt{E}_0^b \\ \Gamma; \Pi \vdash \mathtt{E}_i^s : \mathtt{T}_i \leadsto \mathtt{E}_i^b\ \forall i \in 1,\dots,n \\ \Gamma \vdash \mathtt{C.m}(\mathtt{T}_1 \dots \mathtt{T}_n) \xrightarrow{res} <\bar{\mathtt{T}}', \mathtt{T}'>\end{array}}{\Gamma; \Pi \vdash \mathtt{E}_0^s.m(\mathtt{E}_1^s,\dots,\mathtt{E}_n^s) : \mathtt{T}' \leadsto \mathtt{E}_0^b \ll \mathtt{C.T}'(\bar{\mathtt{T}}') \gg \mathtt{m}(\mathtt{E}_1^b,\dots,\mathtt{E}_n^b)}$$

Figure 3.3: Separate compilation

classes are defined and the Java rules on overriding are respected (that is, a class cannot declare a method with the same name and parameter types of an inherited method and different return type). The rules for well-formed type environments are shown in Figure 3.5.

Rules (*empty*), (*conc*), and (*singleton*) ensure the basic properties expected by an entailment relation. Note that $\Gamma_1 \vdash \Gamma_2$ is provable only if Γ_1 is well-formed.

A class type `C` is defined in Γ if `C` is declared in Γ *def*, whereas types `Object` and `int` are always defined (rules (*Object*) and (*int*)).

Rules for subtyping are standard.

Rule (*exact res*) deals with the situation where there exists a method in a superclass of the

$$(\textit{well-def})\ \frac{\Gamma^{\mathrm{s}} \vdash \Gamma \quad \vdash \Gamma^{\mathrm{s}} \diamond}{\Gamma \diamond}$$

$$(\textit{empty})\ \frac{\Gamma \diamond}{\Gamma \vdash \Lambda} \qquad (\textit{conc})\ \frac{\Gamma \vdash \Gamma_1 \quad \Gamma \vdash \gamma}{\Gamma \vdash \Gamma_1, \gamma} \qquad (\textit{singleton})\ \frac{\Gamma_1, \gamma, \Gamma_2 \diamond}{\Gamma_1, \gamma, \Gamma_2 \vdash \gamma}$$

$$(\textit{def})\ \frac{\Gamma \vdash \mathtt{C} \mapsto <\mathtt{C}', \mathtt{NHS}>}{\Gamma \vdash \exists\, \mathtt{C}} \qquad (\textit{Object})\ \frac{\Gamma \diamond}{\Gamma \vdash \exists\, \mathtt{Object}} \qquad (\textit{int})\ \frac{\Gamma \diamond}{\Gamma \vdash \exists\, \mathtt{int}}$$

$$(\textit{refl})\ \frac{\Gamma \vdash \exists\, \mathtt{T}}{\Gamma \vdash \mathtt{T} \leq \mathtt{T}} \qquad (\textit{trans})\ \frac{\Gamma \vdash \mathtt{C}_1 \leq \mathtt{C}_2 \quad \Gamma \vdash \mathtt{C}_2 \mapsto <\mathtt{C}_3, \mathtt{NHS}>}{\Gamma \vdash \mathtt{C}_1 \leq \mathtt{C}_3}$$

$$(\textit{top})\ \frac{\Gamma \vdash \exists\, \mathtt{C}}{\Gamma \vdash \mathtt{C} \leq \mathtt{Object}} \qquad (\textit{vector})\ \frac{\Gamma \vdash \mathtt{T}_i \leq \mathtt{T}'_i\ \forall\, i \in 1, \ldots, n}{\Gamma \vdash \mathtt{T}_1 \ldots \mathtt{T}_n \leq \mathtt{T}'_1 \ldots \mathtt{T}'_n}$$

$$(\textit{exact res})\ \frac{\Gamma \vdash \mathtt{C}' \mapsto <\mathtt{C}'', \mathtt{NHS}> \quad \Gamma \vdash \mathtt{C} \leq \mathtt{C}'}{\Gamma \vdash \mathtt{C.m}(\bar{\mathtt{T}}) \stackrel{res}{\rightarrow} <\bar{\mathtt{T}}, \mathtt{T}>} \qquad \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{NHS}$$

$$(\textit{match res})\ \frac{\begin{array}{l} \textit{applAll}(\Gamma, \mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s} \\ \textit{matchAll}(\Gamma, \mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s} \\ \textit{mostSpec}(\Gamma, \mu\mathtt{s}) = <\bar{\mathtt{T}}', \mathtt{T}'> \end{array}}{\Gamma \vdash \mathtt{C.m}(\bar{\mathtt{T}}) \stackrel{res}{\rightarrow} <\bar{\mathtt{T}}', \mathtt{T}'>}$$

$$(\textit{complete res})\ \frac{\begin{array}{l} \textit{applAll}(\Gamma, \mathtt{C}, \mathtt{m}, \bar{\mathtt{T}}) = \mu\mathtt{s} \\ \Gamma \vdash \bar{\mathtt{T}} \Uparrow \\ \textit{mostSpec}(\Gamma, \mu\mathtt{s}) = <\bar{\mathtt{T}}', \mathtt{T}'> \end{array}}{\Gamma \vdash \mathtt{C.m}(\bar{\mathtt{T}}) \stackrel{res}{\rightarrow} <\bar{\mathtt{T}}', \mathtt{T}'>}$$

$$(\smiley\, \textit{obj})\ \frac{\Gamma \diamond}{\Gamma \vdash \mathtt{Object} \smiley \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})}$$

$$(\smiley\, \textit{down})\ \frac{\begin{array}{l} \Gamma \vdash \mathtt{C} \mapsto <\mathtt{C}', \{\mathtt{NH}_1, \ldots, \mathtt{NH}_n\}> \\ \Gamma \vdash \mathtt{C}' \smiley \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \end{array}}{\Gamma \vdash \mathtt{C} \smiley \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})} \qquad \begin{array}{l} \mathtt{NH}_i = \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \implies \mathtt{T} = \mathtt{T}' \\ \forall\ i \in 1, \ldots, n \end{array}$$

$$(\textit{not sub})\ \frac{\Gamma \vdash \mathtt{C} \Uparrow \quad \mathtt{C}' \notin \textit{supertypes}(\Gamma, \mathtt{C})}{\Gamma \vdash \mathtt{C} \not\leq \mathtt{C}'}$$

Figure 3.4: Type environments entailment

$$
\frac{\Gamma^{\mathrm{s}} \vdash \Gamma^{\mathrm{s}}\diamond}{\vdash \Gamma^{\mathrm{s}}\diamond}
\qquad
\frac{}{\Gamma^{\mathrm{s}} \vdash \Lambda\diamond}
\qquad
\frac{}{\Gamma^{\mathrm{s}} \vdash \mathtt{T}\diamond_{\mathtt{type}}}
\begin{array}{l}
\mathtt{T} = \mathtt{int}\vee \\
\mathtt{T} = \mathtt{Object}\vee \\
\mathtt{T} \in \mathit{Def}(\Gamma^{\mathrm{s}})
\end{array}
$$

$$
\frac{\Gamma^{\mathrm{s}} \vdash \mathtt{T}_i \diamond_{\mathtt{type}}\ \forall i \in 0, \ldots, n}{\Gamma^{\mathrm{s}} \vdash \mathtt{T}_0\ \mathtt{m}(\mathtt{T}_1..\mathtt{T}_n)\diamond_{\mathtt{NH}}}
\qquad
\frac{\Gamma^{\mathrm{s}} \vdash \mathtt{NH}_i \diamond_{\mathtt{NH}}\ \forall i \in 1, \ldots, n}{\Gamma^{\mathrm{s}} \vdash \mathtt{NH}_1 \ldots \mathtt{NH}_n\diamond_{\mathtt{NHS}}}
$$

$$
\frac{}{\Gamma^{\mathrm{s}} \vdash \mathtt{Object} : \emptyset}
\qquad
\frac{\begin{array}{c}\Gamma^{\mathrm{s}} \vdash \mathtt{C}' : \mathtt{NHS}' \\ \Gamma^{\mathrm{s}} \vdash \mathtt{NHS}\diamond_{\mathtt{NHS}}\end{array}}{\Gamma^{\mathrm{s}} \vdash \mathtt{C} : \mathtt{NHS} \cup \mathtt{NHS}'}
\begin{array}{l}
\Gamma^{\mathrm{s}}(\mathtt{C}) = <\mathtt{C}', \mathtt{NHS}> \\
\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{NHS}, \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{NHS}' \implies \mathtt{T} = \mathtt{T}' \\
\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}}), \mathtt{T}'\ \mathtt{m}(\bar{\mathtt{T}}) \in \mathtt{NHS} \implies \mathtt{T} = \mathtt{T}'
\end{array}
$$

$$
\frac{\Gamma^{\mathrm{s}} \vdash \Gamma^{\mathrm{s}}_1\diamond \quad \Gamma^{\mathrm{s}} \vdash \mathtt{C} : \tau}{\Gamma^{\mathrm{s}} \vdash \Gamma^{\mathrm{s}}_1, \mathtt{C} \mapsto \tau\diamond}
\Gamma^{\mathrm{s}}_1(\mathtt{C}) = \tau' \implies \tau' = \tau
$$

Figure 3.5: Well-formed standard type environments

type of the receiver with parameters of the same type of the arguments; clearly, in this case invocation will be always resolved to that method, despite of the other type assumptions contained in Γ.

Rule (*match res*) requires more type assumptions than the previous rule in order to be applicable: the standard type assumptions of all superclasses of the type `C` of the receiver - `C` included - must be in Γ. Then it is possible to compute all applicable methods (function *applAll*) and to verify that each applicable method is matchable (function *matchAll*) as well. Finally, the set of all applicable methods must contain the most specific method (function *mostSpec*). A method $\mathtt{T}'\ \mathtt{m}'(\bar{\mathtt{T}}')$ is applicable to the invocation $\mathtt{C.m}(\bar{\mathtt{T}})$ in Γ if $\mathtt{m}' = \mathtt{m}$ and $\bar{\mathtt{T}} \leq \bar{\mathtt{T}}'$ can be proved in Γ, while is matchable if $\mathtt{m} = \mathtt{m}'$ and there exists a type environment where $\bar{\mathtt{T}} \leq \bar{\mathtt{T}}'$ can be proved. All these auxiliary functions are defined in Figure 3.6.

Rule (*complete res*) can be applied if the standard type assumptions of all superclasses of the types `C` and $\bar{\mathtt{T}}$ - `C` and $\bar{\mathtt{T}}$ included - of the receiver and of the arguments ($\Gamma \vdash \bar{\mathtt{T}} \Uparrow$) can be found in Γ; in this case the set of all applicable methods is the same in any environment entailed by Γ, therefore there is no need to compute the matchable methods.

Rule (☺*obj*) states that `Object` can be safely extended by any method[2] (that is, without breaking the Java rule on overriding); in all other cases, if a class $\mathtt{C}'$ can be safely extended by a method $\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$ then any direct subclass `C` of $\mathtt{C}'$ can be safely extended by the same method, providing that `C` does not contain a method with the same name, same types of

[2]For simplicity, we ignore all the predefined methods of `Object`, defined in 4.3.2 of [GJSB00].

$$\frac{}{\Gamma \vdash \texttt{Object} \Uparrow} \qquad \frac{\Gamma \vdash \texttt{C}' \Uparrow}{\Gamma \vdash \texttt{C} \Uparrow}\,\Gamma(\texttt{C}) = <\texttt{C}', \_> \qquad \frac{\Gamma \vdash \texttt{T}_1 \Uparrow \ldots \Gamma \vdash \texttt{T}_n \Uparrow}{\Gamma \vdash \texttt{T}_1 \ldots \texttt{T}_n \Uparrow}$$

$$\Gamma \vdash \texttt{int} \Uparrow$$

$$\mathit{supertypes}(\Gamma, \texttt{C}) = \begin{cases} \{\texttt{Object}\} & \text{if } \texttt{C} = \texttt{Object} \\ \{\texttt{C}\} \cup \mathit{supertypes}(\Gamma, \texttt{C}') & \text{if } \Gamma(\texttt{C}) = <\texttt{C}', \_> \\ \bot & \text{otherwise} \end{cases}$$

$$\mathit{Appl}(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \begin{cases} \{\ll \texttt{C.T}\ (\bar{\texttt{T}}') \gg \mid \texttt{T m}(\bar{\texttt{T}}') \in \texttt{NHS}, \Gamma \vdash \bar{\texttt{T}} \leq \bar{\texttt{T}}'\ \} & \text{if } \Gamma(\texttt{C}) = <\_, \texttt{NHS}> \\ \bot & \text{otherwise} \end{cases}$$

$$\mathit{applAll}(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \begin{cases} \emptyset & \text{if } \texttt{C} = \texttt{Object} \\ \mu\texttt{s}_1 \cup \mu\texttt{s}_2 & \text{if } \Gamma(\texttt{C}) = <\texttt{C}', \_> \\ & \quad \mathit{Appl}(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \mu\texttt{s}_1, \\ & \quad \mathit{applAll}(\Gamma, \texttt{C}', \texttt{m}, \bar{\texttt{T}}) = \mu\texttt{s}_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\mathit{match}(\texttt{T}_1 \ldots \texttt{T}_m, \texttt{T}'_1 \ldots \texttt{T}'_n) \iff m = n \wedge \forall i \in 1..n\ (\texttt{T}_i = \texttt{int}) \iff (\texttt{T}'_i = \texttt{int})$$

$$\mathit{match}(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \begin{cases} \{\ll \texttt{C.T}\ (\bar{\texttt{T}}') \gg \mid \texttt{T m}(\bar{\texttt{T}}') \in \texttt{NHS}, \mathit{match}(\bar{\texttt{T}}, \bar{\texttt{T}}')\} & \text{if } \Gamma(\texttt{C}) = <\_, \texttt{NHS}> \\ \bot & \text{otherwise} \end{cases}$$

$$\mathit{matchAll}(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \begin{cases} \emptyset & \text{if } \texttt{C} = \texttt{Object} \\ \mu\texttt{s}_1 \cup \mu\texttt{s}_2 & \text{if } \Gamma(\texttt{C}) = <\texttt{C}', \_> \\ & \quad \mathit{match}(\Gamma, \texttt{C}, \texttt{m}, \bar{\texttt{T}}) = \mu\texttt{s}_1, \\ & \quad \mathit{matchAll}(\Gamma, \texttt{C}', \texttt{m}, \bar{\texttt{T}}) = \mu\texttt{s}_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\mathit{mostSpec}(\Gamma, \mu\texttt{s}) = \begin{cases} \ll \texttt{C.T}\ (\bar{\texttt{T}}) \gg & \text{if } \ll \texttt{C.T}\ (\bar{\texttt{T}}) \gg \in \mu\texttt{s} \text{ and} \\ & \quad \Gamma \vdash \texttt{C} \leq \texttt{C}', \\ & \quad \Gamma \vdash \bar{\texttt{T}} \leq \bar{\texttt{T}}' \text{ for all } \ll \texttt{C}'.\texttt{T}'\ (\bar{\texttt{T}}') \gg \in \mu\texttt{s} \\ \bot & \text{otherwise} \end{cases}$$

Figure 3.6: Auxiliary judgment and functions

parameters but *different* return type (rule ($\smiley$ *down*)).

Finally, rule (*not sub*) is applicable only if Γ contains the standard type assumptions of all supertypes of $\mathtt{C}$ - $\mathtt{C}$ included ($\Gamma \vdash \mathtt{C} \Uparrow$) - and $\mathtt{C}'$ is not in the set of such supertypes (function *supertypes*).

3.3 Selective recompilation

Despite several papers have been written on the subject of selective recompilation, Dmitriev's approach [Dmi02] is the only other Java specific proposal we are aware of. Dmitriev's paper describes a make technology, based on smart dependency checking, that aims to keep a project (that is, a set of source and binary fragments) consistent while reducing the number of files to be recompiled. A project is said to be consistent when all its sources can be recompiled producing the same binaries as before. The main idea is to catalog all possible changes to a source code (as, for instance, adding/removing methods) establishing a criterion for finding a subset of dependent classes that have to be recompiled. A freely downloadable tool, *Javamake*, is based on such a paper and implements the selective recompilation for Java upon any Java compiler. This tool stores some type information for each project in database files which are used to determine which changes have been made to the sources with respect to the previous (consistent) version. Even though this approach has the advantage of being the only one to be both well documented and fully implemented, unfortunately, is not based on a theoretical foundation, as pointed out by the author himself. As a consequence, no proof of correctness is provided, therefore there is no guarantee that *Javamake* always forces the recompilation of a class when needed for ensuring the consistency of the project. Furthermore, *Javamake* cannot avoid a considerable amount of unnecessary recompilations.

Instead, the type system given in this chapter provides a good theoretical basis for implementing selective recompilation strategy. Consider, for instance, to extract a (standard) type environment $\Gamma^{\mathrm{S}}_{\mathtt{old}}$ from a program $\mathtt{P}$ and to compile one of its sources $\mathtt{S}$:

$$\Gamma^{\mathrm{S}}_{\mathtt{old}} \vdash \mathtt{S} : \_ \leadsto \mathtt{B}$$

The proof tree for this judgment contains a set of type assumptions $\Gamma^{\mathrm{NS}} = \gamma_1^{\mathrm{NS}}, \ldots, \gamma_n^{\mathrm{NS}}$. We call this set the *requirements* for $\mathtt{S}$ in $\Gamma^{\mathrm{S}}_{\mathtt{old}}$ and write

$$\mathit{Reqs}(\mathtt{S}, \Gamma^{\mathrm{S}}_{\mathtt{old}}) = \Gamma^{\mathrm{NS}}$$

Assume now to change the program $\mathtt{P}$, leaving the fragment $\mathtt{S}$ untouched. Does $\mathtt{S}$ compile in the new type environment $\Gamma^{\mathrm{S}}_{\mathtt{new}}$ (extracted from the updated program)? If it does, can we say something about the corresponding binary?

If the requirements for S describe exactly the *weakest* assumptions in order to compile S in B, then, as long as $\Gamma^{\mathrm{S}}_{\mathtt{new}} \vdash \Gamma^{\mathrm{NS}}$ we know that a recompilation of S would produce the *same*, existing, binary B and, conversely, that $\Gamma^{\mathrm{S}}_{\mathtt{new}} \not\vdash \Gamma^{\mathrm{NS}}$ implies that S cannot be compiled into the same binary B.

This turns out to be true: indeed, during the compilation of a source S to a binary B, it is possible to infer the weakest type assumptions Γ^{NS} needed for compiling S to B. Then, after some changes, as long as the new standard environment entails Γ^{NS}, there is no need to recompile the source as its recompilation would produce the same binary. We omit the proof for the subset of Java modeled so far, since Theorem 4.2.9, in the next chapter, proves this important property for a larger subset of Java. That proof ensures not only that this approach is correct, but also that it is minimal in the following sense: whenever a class is recompiled, either the recompilation fails, or it generates a different binary.

Let us sketch the algorithm here, to show the idea on the the example already discussed in Section 3.1:

```
class P extends Object {
 int f(Object o) { return 0 ; }
 // int f(int i) { return i ; }
}
class U extends Object {}
class Y extends X {}
class X extends Object {}

class H extends P {
 int g(P p) { return p.f(new(H)) ; }
 int m()    { return new H().g(new P()) ; }
 U id(U u)  { return u ; }
 X em(Y y)  { return y ; }
}
```

These classes compile successfully, and they form our example project. The compiler would generate the following weakest environment for H:

$$\begin{aligned}\Gamma^{\mathrm{NS}} = {} & \exists\,\mathtt{P}, \exists\,\mathtt{U}, \exists\,\mathtt{X}, \exists\,\mathtt{Y}, \mathtt{P} \not\leq \mathtt{H}, \mathtt{P}☺\mathtt{int}\ \mathtt{g}(\mathtt{P}), \mathtt{P}☺\mathtt{int}\ \mathtt{m}(),\\ & \mathtt{P}☺\mathtt{U}\ \mathtt{id}(\mathtt{U}), \mathtt{P}☺\mathtt{X}\ \mathtt{em}(\mathtt{Y}), \mathtt{P.f}(\mathtt{H}) \overset{\mathtt{res}}{\to} <\mathtt{Object}, \mathtt{int}>,\\ & \mathtt{H.g}(\mathtt{P}) \overset{\mathtt{res}}{\to} <\mathtt{P}, \mathtt{int}>, \mathtt{Y} \leq \mathtt{X}\end{aligned}$$

If we add a method `f(int)` in P, then class H still invokes the same method as before, because the new method is not even applicable to the invocation with an argument of type P. These considerations are formally captured by the entailment relation: the new standard environment (that can be extracted from the new source for P and the old binary of H) still entails Γ^{NS} therefore there is no need to recompile H. For instance, the reader can verify that $\texttt{P.f(H)} \stackrel{\texttt{res}}{\to} \texttt{<Object,int>}$ can still be entailed. On the other hand, whereas *Javamake*[3] is able to detect that classes U, X and Y need not to be recompiled, since they do not use P at all, it cannot distinguish between changes to a set of overloaded methods that alter the resolution of a particular invocation and changes that do not. So, *Javamake* would unnecessarily recompile class H, because it contains an invocation to P.f, producing the same binary as before.

Our whole algorithm can be sketched as follows:

let $S = \{i \mid 1 \leq i \leq n \text{ and } F_i \text{ has been modified}\}$
let $T = \{1, \ldots, n\} \setminus S$
let $\Gamma^{\text{S}} = \mathit{extract}(\{\texttt{S}_i \mid i \in S\} \cup \{\texttt{B}_j \mid j \in T\})$
for all $j \in T$ if not $\mathit{entails}(\Gamma^{\text{S}}, \Gamma^{\text{NS}}_j)$ then $S = S \cup \{j\}$
for all $i \in S$ if $\mathit{compile}(\Gamma^{\text{S}}, \texttt{S}_i) = \mathit{fail}$ then fail
else $(\texttt{B}_i, \Gamma^{\text{NS}}_i) = \mathit{compile}(\Gamma^{\text{S}}, \texttt{S}_i)$

where n is the number of fragments, F_i, $\texttt{S}_i$, $\texttt{B}_i$ and Γ^{NS}_i denote the name, source, binary and weakest assumptions of the i-th fragment, respectively. The function *extract* extracts the corresponding standard type environment from a set of sources and binaries, and *entails* and *compile* implement the entailment relation and the compilation judgment of the typing system, respectively. However, differently from what happens in the system, both *entails* and *compile* do not need to check that the environment Γ^{S} is well-formed, indeed it can be proved that if $\mathit{entails}(\Gamma^{\text{S}}, \Gamma^{\text{NS}}_i) = \mathit{true}$ and $\mathit{compile}(\Gamma^{\text{S}}, \texttt{S}_i) = (\texttt{B}_i, \Gamma_i)$ for all $i \in 1, \ldots, n$, then Γ^{S} is well-formed. Finally, $\mathit{compile}(\Gamma^{\text{S}}, \texttt{S}_i)$ returns, besides the binary $\texttt{B}_i$, also the weakest environment Γ^{NS}_i s.t. $\texttt{S}_i$ compiles to $\texttt{B}_i$ (hence, by definition $\mathit{entails}(\Gamma^{\text{S}}, \Gamma^{\text{NS}}_i) = \mathit{true}$).

While the idea is simple, extending this approach to the full Java language is quite challenging, because some apparently orthogonal features of Java interact badly with separate compilation, as we discuss in the next chapter.

[3]Version 1.3.1, the latest available when we run this test.

Chapter 4

Towards selective recompilation for full Java

In the previous chapter we have presented a recompilation strategy which is optimal in the sense that an unchanged fragment is recompiled if and only if its recompilation produces a different result than the previous compilation (either a new binary or an error). However, the subset of Java we have modeled so far does not include some Java-specific features which badly interact with selective recompilation. Therefore, in this chapter we extend the previous type system to these features in order to provide a solid basis for implementing a compiler manager based on these ideas [Lag04b].

In Section 4.1, by means of some examples, we analyze the various kind of dependencies that may be present among fragments and how to model them. In Section 4.2 we present the extended type system and prove it can be used to obtain an optimal recompilation strategy. Finally, in Section 4.3 we discuss some implementation issues.

4.1 An informal overview

In the previous chapter we have shown some examples of how method overloading interacts with selective recompilation. In those example, however, we did not take into account the accessibility of members; that is, the fact that different clients have different visibility of the same class `C`. Accessibility affects selective recompilation too, as the following example shows.

Consider, for instance, to compile (successfully) the following declarations, assuming each declaration to be in a separate file named after the class.

```
class Client {
    void f(C c) {
        c.m(c) ;
    }
}

class C {
    void m(Object o) {}
}
```

If we added, for instance, a method `m(C)` to class `C` as `private`, then the most specific method seen by `Client` would be still `m(Object)` and a recompilation of `Client` would be useless, since it would produce a binary fragment equal to the existing one.

Another feature that affects the generated code, corresponding to a method invocation, is the kind of the most specific method: an invocation to an instance method is translated into the JVM instruction `invokevirtual`, while an invocation of a static one into `invokestatic`.

For these reasons, we need to extend the type assumption which describes the most specific method for a method invocation as follows:

$$\mathtt{C} \not< \mathit{Mth}(\mathtt{C}', \mathtt{m}, \bar{\mathtt{T}}^{\perp}) = [\textsc{mk}{=}\mathtt{MK}^{\star}, \textsc{ret}{=}\mathtt{T}, \textsc{par}{=}\bar{\mathtt{T}}]$$

with the informal meaning (the details are given in Section 4.2) that if some code contained in class $\mathtt{C}$ invokes a method named $\mathtt{m}$ on an object of class $\mathtt{C}'$ with arguments of type $\bar{\mathtt{T}}^{\perp}$, then the most specific method must be $\mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})$ with kind $\mathtt{MK}^{\star}$ — $\bar{\mathtt{T}}^{\perp}$ is a sequence of types which may contain the type of `null` and $\mathtt{MK}^{\star}$ specifies the kind of the method, that is, whether it is an instance or a static method.

The assumption for the specific example is:

$$\mathtt{Client} \not< \mathit{Mth}(\mathtt{C}, \mathtt{m}, \mathtt{C}) = [\textsc{mk}{=}\epsilon, \textsc{ret}{=}\mathtt{int}, \textsc{par}{=}\mathtt{Object}]$$

This means that the bytecode corresponding to the invocation inside class `Client` remains the same as long as the most specific method:

- is an instance method ($\mathtt{MK} = \epsilon$, as opposed to static methods with $\mathtt{MK} = \mathtt{static}$);
- receives an `Object`;
- returns an `int`.

Note that in a different type environment the most specific method could be different (for instance, it could be found in a different class), but as long as the type assumption hold, that is, the most specific method matches all the characteristics listed above, the bytecode corresponding to the method invocation does not change.

A similar reasoning applies to exception specifications (that is, `throws` clauses) of methods; as the reader may have noted, there is no mention of exception specifications in the method invocation assumptions. Of course, declared exceptions have to be taken into account when typechecking, but they must be handled by their own type assumptions[1] (as discussed below). Indeed, a change to the exception specification, of the most specific method for a given invocation, does not imply, by itself, that such an invocation is now incorrect (note that a change to the exception specification *never* affect the generated bytecode, but it might make the code containing the invocation illegal because it throws an undeclared exception or because a piece of code becomes unreachable; both these events are discussed below in their own sections).

4.1.1 Compile-time constant fields

Compile-time constant fields are `static final` fields, of a primitive type or `String`, which are initialized by a compile-time constant. For the sake of brevity we will call them *ctc-fields* from now on. Because the value of these fields is a compile-time constant (see[2] JLS 15.28), the bytecode generated for an access to a ctc-field is the same that would be generated if the literal corresponding to the value of the field was used in its place.

Consider, for instance, the following declarations (assuming, as before, that each class is declared in a separate file).

```
class A {
    static final int CONST_A = 1 ;
}

class B {
    static final int CONST_B = 1 ;
}
```

[1] In our previous work on the subject [Lag03, Lag04a] method invocation assumptions *did* contain exception specifications. The idea was to specify the "maximum" set of allowed exceptions; while this approach works, it is not expressive enough to model unreachable code and it has become redundant with exception type assumptions we now use.

[2] From now on JLS stands for a reference to the the Java Language Specification [GJSB00].

```
class FirstClient {
    int ma() {
        return A.CONST_A ;
    }
    int mb() {
        return B.CONST_B ;
    }
}

class SecondClient {
    int m() {
        return A.CONST_A+B.CONST_B ;
    }
}
```

These sources are compiled to the following binaries[3]:

```
class A {
    static final int CONST_A = 1 ;
}
class B {
    static final int CONST_B =1 ;
}
class FirstClient {
    int ma() { return 1 ; }
    int mb() { return 1 ; }
}
class SecondClient {
    int m() { return 2 ; }
}
```

As shown in the example, in the binaries every symbolic reference to ctc-fields has disappeared. Indeed, there is no need to calculate at run-time a value which is a compile-time constant. This means that every time the value of a ctc-field is changed and the class where it has been declared is recompiled, its clients might need to be recompiled as well, in order to have a sound compilation strategy. Indeed, the bytecode corresponding to the method `ma` declared in class `FirstClient` changes each time the value of `A.CONST_A` changes. So, we need to keep track of the dependencies between the definition of ctc-fields and their uses. However, the naive approach of using assumptions of the form "field name=value" is sound

[3]As in previous chapters, we use a very abstract view of the bytecode. In a real `.class` file the initializer expressions for all ctc-fields are not present (JLS 13.4.8) and their values are stored as attributes of type `ConstantValue` (see 4.7.2 of [LY99]).

but not minimal. As the example shows, client classes `FirstClient` and `SecondClient` use both constants, but the former needs both `A.CONST_A` and `B.CONST_B` to be equal to `1` to be recompiled to the same bytecode, while the latter just needs their sum to be equal to `2`. Thus, if we change, say, `CONST_A` to `0` and `CONST_B` to `2`, then the first client has to be recompiled, while the second has not. Indeed, after the change, the expression `A.CONST_A+B.CONST_B` has still the same value it had before, so class `SecondClient` would be recompiled to the same existing binary. Hence, to achieve minimality we use type assumptions of the form "expression=value" where the expression is built on values and references to fields (which must be constant). Moreover, we also need to keep track of the fact that a certain accessed field `f` *is not* a ctc-field. This is needed because every access to `f` is compiled to a field-access instruction in the bytecode but, if `f` became a ctc-field, then the compile-time value of `f` would be directly used instead. In our framework this is modeled by two judgments whose derivation is mutually exclusive, as we detail in Section 4.2.

Note that, in this case, the generated bytecode is different, even though its semantics is the same (fetching the value of a ctc-field has the same result of using directly its value), except for timing issues. However, these timing issues might introduce subtle bugs in multithreaded environments; for this reason, we prefer to stick to the choice of recompiling a fragment each time the produced bytecode is different from the existing one.

Note that compiling a single source expression may lead to several assumptions; for instance, the source expression `A.CONST_A+aMethod(A.CONST_A>=B.CONST_B)`, is compiled[4] to `1+aMethod(true)` and generates two type assumptions:

1. `A.CONST_A=1` and
2. `A.CONST_A>=B.CONST_B` = `true`.

These assumptions model the fact that this source expression is compiled to the same bytecode whenever `A.CONST_A` is equal to `1` (because this is the value of the left operand of the sum) and `B.CONST_B` is any integer less or equal to `1` (that is, the value `A.CONST_A` is constrained to be by the first assumption). In some cases these assumptions could be simplified; for instance, the second one can be simplified, as just noted, to `1>=B.CONST_B` = `true`. In the general case, anyway, this would require an analysis of assumptions deduced from different expressions, which we think it is not worth the effort.

[4]Ignoring method invocation annotations, which are of no interest in the context of this example.

4.1.2 Unreachable code

The way Java handles unreachable code is peculiar: what is a warning in most languages is a compile-time error in Java, JLS 14.20. The relation between this choice and selective recompilation lies in the fact that a change in some source may make unreachable a piece of code, contained in an unchanged source, which was reachable before. So, a source which was perfectly legal is not correct anymore. In this case, a sound recompilation strategy must trigger the compilation on that source, in order to raise the error (unreachable code) that a global recompilation would raise.

Two ways to make unreachable a piece of code, which was reachable, without changing its source are: changing the value of a ctc-field and changing the exception specification of a method. Examples of both cases are shown below, starting with the simpler one: changing the value of a ctc-field (making it a member of the infamous "inconstant constants[5]" club ☺). For instance, consider:

```
class ThirdClient {
    void m() {
        int x = 0 ;
        while (A.CONST_A==B.CONST_B)
            ++x ;
    }
}
```

This client[6] class is successfully compiled only when both ctc-fields `A.CONST_A` and `B.CONST_B` have the same value; indeed, when they do not, the increment of `x` can never be reached.

The following example shows that changing an exception specification can do the trick too.

```
class E extends Exception { ... }
class Foo {
    Foo m() throws E { ... }
    Foo m2() { ... }
}
class Client {
    ...
    try   {
        new Foo().m().m2() ;
    } catch (E e) {
        ++x ;
    }
```

[5]JLS 13.4.8.

[6]of classes `A` and `B` which have been defined in Section 4.1.1.

```
    ...
}
```

If `E` is removed from the exception specification of method `m`, then the increment of `x` inside the `catch` becomes unreachable. As for ctc-fields, we introduce a new assumption to model the requirements which guarantee a piece of code to be reachable. Again, a naive assumption like "method `m` throws `E`" is not enough to obtain a minimal, although sound, compilation strategy: indeed, if we move `E` from the exception specification of `m` to the exception specification of `m2`, then the code inside the `catch` remains reachable. The nesting of try statements complicates the matter further; consider, for instance, the following code:

```
try {
    m1() ;
    try {
        m2() ;
    } catch (E1 e1) { }
} catch (E2 e2) { ++x ; }
```

In this example the increment of `x` is reachable if and only if:

- method `m1` has an exception specification which includes a subclass of `E2` (which can, of course, be `E2` itself) *or*
- method `m2` has an exception specification which includes a subclass of `E2` which is not a subclass of `E1` (otherwise such an exception would be captured by the `catch` clause of the inner `try`).

More precisely, assumptions like those above are not related to a fixed method, say `m1`, but to the most specific method for the invocation. Indeed, an invocation like "`m1()`" may throw all the exceptions which are specified in the throw clause of the *most specific method* named `m1`, without parameters, on an object of type `Foo` as seen by class `Client`.

Using $\varepsilon(\texttt{Client} \not\lessdot \texttt{Foo.m()})$ to indicate the exception specification of the most specific method for an invocation of `m()` on a receiver of type `Foo` inside class `Client`, we can express the above assumptions as shown below:

$$(\ \varepsilon(\texttt{Client} \not\lessdot \texttt{Foo.m1()})\ \cup_{\texttt{Exc}}\ (\varepsilon(\texttt{Client} \not\lessdot \texttt{Foo.m2()}) \setminus_{\texttt{Exc}} \{\texttt{E1}\})\ )\ \supseteq_{\texttt{Exc}} \{\texttt{E2}\}$$

The operators $\cup_{\texttt{Exc}}$, $\setminus_{\texttt{Exc}}$ and $\supseteq_{\texttt{Exc}}$ are similar to set union, minus and containment, but they take also in account the exception hierarchy. These operators are discussed in the next section.

4.2 Formalization

In this section we formalize the ideas presented so far. Section 4.2.1 describes the subset of Java we model, both at source and binary level. Section 4.2.2 introduces type environments and describes how these can be extracted from fragments. Section 4.2.3 lists all the type assumptions which model the fine-grained requirements which needs to be checked both to compile a source fragment and to decide whether it has to be recompiled after some changes have been made. Section 4.2.4 presents the typing rules. Section 4.2.5 proves the type system can used to provide a sound and minimal compilation strategy. Finally, Section 4.2.6 show how type environments can be checked incrementally.

4.2.1 The language

In this chapter we model a substantial subset of Java at both source (Figure 4.1) and binary (Figure 4.2) level. As before, our model of bytecode is rather abstract: it is basically source code enriched with some annotations (discussed below). However, this time, the same source level expression can be compiled to different kinds of binary level expressions, as it happens in Java. For instance, a method invocation `x.m()` can be translated to a virtual method invocation or an interface method invocation depending on the static type of `x`.

With the exception of arrays and inner-classes, we model all the major features of Java: classes (including abstract classes), interfaces, primitive types, access modifiers (including packages, but without the `import` directive), constructors, (instance/static) fields (both in classes and interfaces), (instance/static/abstract) methods, super field accesses and method invocations, exceptions. The treatment of arrays and inner-classes would complicate the model without apparently giving further insights. In the few points where inner-classes would make a difference we briefly discuss the issue.

Figure 4.1 gives the syntax of the source language. A source fragment `S` can be a class declaration or an interface declaration. The former consists of: an access modifier `AM`, a class kind `CK` (either ϵ or `abstract`), the name of the class, the name of the superclass, the list of the implemented interfaces and the declaration of constructors, fields and methods. Analogously, an interface declaration consists of an access modifier, the name of the interface, the names of the superinterfaces and the declaration of fields and methods.

A constructor declaration $\texttt{KD}^s$ consists of an access modifier `AM`, a constructor header `KH`, the invocation of a superclass's constructor[7] and a sequence of statements $\texttt{STMTS}^s$. A constructor header consists of the sequence of the parameters with their types and the

[7]Invocations of a constructor of the same class (using `this`) are not considered since they are simply syntactic shortcuts – JLS 8.8.5.

$$
\begin{array}{rcl}
\mathtt{S} & ::= & \mathtt{AM}\ \mathtt{CK}\ \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{C'}\ \mathtt{implements}\ \mathtt{I}_1 \ldots \mathtt{I}_n\ \{\ \mathtt{KDS}^s\ \mathtt{FDS}^s\ \mathtt{MDS}^s\ \} \mid \\
 & & \mathtt{AM}\ \mathtt{interface}\ \mathtt{I}\ \mathtt{extends}\ \mathtt{I}_1 \ldots \mathtt{I}_n\ \{\ \mathtt{FDS}^s\ \mathtt{MDS}^s\ \} \\
\mathtt{AM} & ::= & \mathtt{public} \mid \mathtt{protected} \mid \epsilon \mid \mathtt{private} \\
\mathtt{CK} & ::= & \epsilon \mid \mathtt{abstract} \\
\mathtt{KDS}^s & ::= & \mathtt{KD}^s_1 \ldots\ \mathtt{KD}^s_n \\
\mathtt{FDS}^s & ::= & \mathtt{FD}^s_1 \ldots\ \mathtt{FD}^s_n \\
\mathtt{MDS}^s & ::= & \mathtt{MD}^s_1 \ldots\ \mathtt{MD}^s_n \\
\mathtt{KD}^s & ::= & \mathtt{AM}\ \mathtt{KH}\ \{\ \mathtt{super}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n);\ \mathtt{STMTS}^s\ \} \\
\mathtt{FD}^s & ::= & \mathtt{AM}\ \mathtt{FINAL}\ \mathtt{FK}\ \mathtt{T}\ \mathtt{f} = \mathtt{E}^s\ ; \\
\mathtt{MD}^s & ::= & \mathtt{AM}\ \mathtt{MK}\ \mathtt{MH}\ \{\ \mathtt{STMTS}^s\ \mathtt{return}\ \mathtt{E}^s;\ \} \mid \mathtt{AM}\ \mathtt{abstract}\ \mathtt{MH}\ ; \\
\mathtt{KH} & ::= & (\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws}\ \mathtt{ES} \\
\mathtt{MK} & ::= & \epsilon \mid \mathtt{static} \mid \mathtt{abstract} \\
\mathtt{MH} & ::= & \mathtt{T}\ \mathtt{m}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws}\ \mathtt{ES} \\
\mathtt{FINAL} & ::= & \epsilon \mid \mathtt{final} \\
\mathtt{FK} & ::= & \epsilon \mid \mathtt{static} \\
\mathtt{T} & ::= & \mathtt{RT} \mid \mathtt{int} \mid \mathtt{bool} \\
\mathtt{RT} & ::= & \mathtt{C} \mid \mathtt{I} \\
\mathtt{ES} & ::= & \{\mathtt{C}_1, \ldots, \mathtt{C}_n\} \\
\mathtt{E}^s & ::= & \mathtt{PRIMARY}^s \mid \mathtt{ASSIGN}^s \mid \nu \mid \mathtt{E}^s_1 + \mathtt{E}^s_2 \mid \mathtt{E}^s_1 - \mathtt{E}^s_2 \mid \ldots \\
\nu & ::= & \beta \mid \iota \\
\beta & ::= & \mathtt{true} \mid \mathtt{false} \\
\iota & ::= & 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots \\
\mathtt{PRIMARY}^s & ::= & \mathtt{null} \mid \mathtt{this} \mid \mathtt{NEW}^s \mid \mathtt{x} \mid \mathtt{INVOKE}^s \mid \mathtt{super.f} \mid \mathtt{PRIMARY}^s.\mathtt{f} \mid \mathtt{RT.f} \\
\mathtt{ASSIGN}^s & ::= & \mathtt{x} = \mathtt{E}^s \mid \mathtt{PRIMARY}^s.\mathtt{f} = \mathtt{E}^s \mid \mathtt{super.f} = \mathtt{E}^s \mid \mathtt{RT.f} = \mathtt{E}^s \\
\mathtt{NEW}^s & ::= & \mathtt{new}\ \mathtt{C}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) \\
\mathtt{INVOKE}^s & ::= & \mathtt{PRIMARY}^s.\mathtt{m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) \mid \mathtt{super.m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) \mid \mathtt{C.m}(\mathtt{E}^s_1, \ldots, \mathtt{E}^s_n) \\
\mathtt{STMTS}^s & ::= & \mathtt{STMT}^s_1 \ldots \mathtt{STMT}^s_n \\
\mathtt{STMT}^s & ::= & \{\mathtt{STMTS}^s\} \mid \mathtt{SE}^s\ ; \mid \mathtt{if}\ (\mathtt{E}^s)\ \mathtt{STMT}^s_1\ \mathtt{else}\ \mathtt{STMT}^s_2 \mid \mathtt{while}\ (\mathtt{E}^s)\ \mathtt{STMT}^s \mid \\
 & & \mathtt{try}\ \{\mathtt{STMTS}^s\ \}\ \mathtt{CATCHES}^s\ \mathtt{finally}\ \{\ \mathtt{STMTS}^s_1\ \} \mid \mathtt{throw}\ \mathtt{E}^s\ ; \mid \mathtt{break}\ ; \\
\mathtt{SE}^s & ::= & \mathtt{ASSIGN}^s \mid \mathtt{INVOKE}^s \mid \mathtt{NEW}^s \\
\mathtt{CATCHES}^s & ::= & \mathtt{CATCH}^s_1 \ldots \mathtt{CATCH}^s_n \\
\mathtt{CATCH}^s & ::= & \mathtt{catch}\ (\mathtt{C}\ \mathtt{x})\ \{\ \mathtt{STMTS}^s\ \}
\end{array}
$$

Assumptions:

- interface names in S are distinct;
- field names in $\mathtt{FDS}^s$ are distinct;
- method (constructor) name/parameters (parameters) in $\mathtt{MDS}^s$ ($\mathtt{KDS}^s$) are distinct;
- parameter and exception names in both KH and MH are distinct;
- class names in $\mathtt{CATCHES}^s$ are distinct.

Figure 4.1: Syntax - Sources

exception specification ES. We assume for simplicity that any class can be an exception, that is, we do not model the predefined class Throwable. So, an exception specification is just a sequence of class names.

A field declaration $\mathtt{FD}^s$ consists of an access modifier AM, a final modifier FINAL, a field kind FK, a type T, the name of the field f and the initialization expression $\mathtt{E}^s$ (for simplicity we assume an initialization expression to be always present).

A method declaration $\mathtt{MD}^s$ can be either concrete or abstract. In the former case it consists of an access modifier AM, a method kind MK, a method header MH, a sequence of statements $\mathtt{STMTS}^s$ and a return expression $\mathtt{E}^s$ (for simplicity we do not model void methods). In the latter case it just consists of an access modifier AM, the keyword abstract and a method header. A type T can be a reference type RT or a primitive type (int or bool). We distinguish between class names C and interface names I for clarity, even though they actually range over the same set of names.

An expression $\mathtt{E}^s$ can be: a primary expression, an assignment expression, a value ν (either a boolean literal β or an integer literal ι) or a binary expression. We show just two examples of arithmetic expressions; they only matter since values can be composed to obtain other values; the syntax for composing them is immaterial. Some expressions, $\mathtt{SE}^s$, can be used as statements; they are: assignment $\mathtt{ASSIGN}^s$, method invocation $\mathtt{INVOKE}^s$ and instance creation $\mathtt{NEW}^s$.

While Java permits accessing a static member of a class/interface RT via both the type name RT or any expression which has static type RT, here we allow only the former kind of access (because allowing both kinds of access would require additional, uninteresting, typing rules).

A statement $\mathtt{STMT}^s$ can be: a block (that is, a sequence of statements $\mathtt{STMTS}^s$ between curly braces), a statement expression (followed by ";"), a conditional if, a while loop, a try-catch-finally block, the throw of an exception, and break. These are a strict subset of what Java offers, but they are enough to model all the issues related to selective recompilation.

Figure 4.2 gives the syntax of the binary language. As already said, it mostly mimics the source language, except for it is enriched with some annotations enclosed between "$\ll$" and "$\gg$". Annotations are:

- *Constructors.* Each invocation of a constructor (via super or new expression) is annotated with $\ll \mathtt{C}(\bar{\mathtt{T}}) \gg_{\mathtt{c}}$, where $\bar{\mathtt{T}}$ specifies the parameter types of the constructor of class C to be invoked. Note that, for mimicking the real translation to the JVM instruction invokespecial, the annotation must specify the class name even when compiling super invocations.

$$
\begin{array}{rcl}
\texttt{B} & ::= & \texttt{AM CK class C extends C}' \texttt{ implements I}_1 \ldots \texttt{I}_n \; \{ \; \texttt{KDS}^b \; \texttt{FDS}^b \; \texttt{MDS}^b \; \} \mid \\
 & & \texttt{AM interface I extends I}_1 \ldots \texttt{I}_n \; \{ \; \texttt{FDS}^b \; \texttt{MDS}^b \; \} \\
\texttt{KDS}^b & ::= & \texttt{KD}^b_1 \ldots \; \texttt{KD}^b_n \\
\texttt{MDS}^b & ::= & \texttt{MD}^b_1 \ldots \; \texttt{MD}^b_n \\
\texttt{FDS}^b & ::= & \texttt{FD}^b_1 \ldots \; \texttt{FD}^b_n \\
\texttt{KD}^b & ::= & \texttt{AM KH} \; \{ \; \ll \texttt{C}(\bar{\texttt{T}}) \gg_{\texttt{c}} (\texttt{E}^b_1, \ldots, \texttt{E}^b_n); \; \texttt{STMTS}^b \; \} \\
\texttt{FD}^b & ::= & \texttt{AM FINAL FK T f} = \texttt{E}^b \\
\texttt{MD}^b & ::= & \texttt{AM MK MH} \; \{ \; \texttt{STMTS}^b \; \texttt{return E}^b; \; \} \mid \texttt{AM abstract MH} \; ; \\
\texttt{E}^b & ::= & \texttt{PRIMARY}^b \mid \texttt{ASSIGN}^b \mid \texttt{N} \mid \texttt{true} \mid \texttt{false} \mid \\
 & & \ldots \mid \texttt{E}^b_1 + \texttt{E}^b_2 \mid \texttt{E}^b_1 - \texttt{E}^b_2 \mid \ldots \\
\texttt{PRIMARY}^b & ::= & \texttt{null} \mid \texttt{this} \mid \texttt{NEW}^b \mid \texttt{x} \mid \texttt{INVOKE}^b \\
 & & \texttt{PRIMARY}^b. \ll \texttt{C.T} \gg_{\texttt{if}} \texttt{f} \mid \ll \texttt{RT.T} \gg_{\texttt{sf}} \texttt{f} \\
\texttt{ASSIGN}^b & ::= & \texttt{x} = \texttt{E}^b \mid \texttt{PRIMARY}^b. \ll \texttt{C.T} \gg_{\texttt{if}} \texttt{f} = \texttt{E}^b \mid \ll \texttt{RT.T} \gg_{\texttt{sf}} \texttt{f} = \texttt{E}^b \\
\texttt{NEW}^b & ::= & \texttt{new} \; \ll \texttt{C}(\bar{\texttt{T}}) \gg_{\texttt{c}} (\texttt{E}^b_1, \ldots, \texttt{E}^b_n) \\
\texttt{INVOKE}^b & ::= & \texttt{PRIMARY}^b. \ll \texttt{C.T} \; (\bar{\texttt{T}}) \gg_{\texttt{vrt}} \texttt{m}(\texttt{E}^b_1, \ldots, \texttt{E}^b_n) \mid \\
 & & \ll \texttt{C.T} \; (\bar{\texttt{T}}) \gg_{\texttt{spr}} \texttt{m}(\texttt{E}^b_1, \ldots, \texttt{E}^b_n) \mid \\
 & & \ll \texttt{C.T} \; (\bar{\texttt{T}}) \gg_{\texttt{stt}} \texttt{m}(\texttt{E}^b_1, \ldots, \texttt{E}^b_n) \mid \\
 & & \texttt{PRIMARY}^b. \ll \texttt{I.T} \; (\bar{\texttt{T}}) \gg_{\texttt{int}} \texttt{m}(\texttt{E}^b_1, \ldots, \texttt{E}^b_n) \\
\texttt{STMTS}^b & ::= & \texttt{STMT}^b_1 \ldots \texttt{STMT}^b_n \\
\texttt{STMT}^b & ::= & \{\texttt{STMTS}^b\} \mid \texttt{SE}^b \; ; \mid \texttt{if} \; (\texttt{E}^b) \; \texttt{STMT}^b_1 \; \texttt{else} \; \texttt{STMT}^b_2 \mid \texttt{while} \; (\texttt{E}^b) \; \texttt{STMT}^b \\
 & & \texttt{try} \; \{\texttt{STMTS}^b \; \} \; \texttt{CATCHES}^b \; \texttt{finally} \; \{ \; \texttt{STMTS}^b_1 \; \} \mid \texttt{throw E}^b \; ; \mid \texttt{break} \; ; \\
\texttt{SE}^b & ::= & \texttt{ASSIGN}^b \mid \texttt{INVOKE}^b \mid \texttt{NEW}^b \\
\texttt{CATCHES}^b & ::= & \texttt{CATCH}^b_1 \ldots \texttt{CATCH}^b_n \\
\texttt{CATCH}^b & ::= & \texttt{catch} \; (\texttt{C x}) \; \{ \; \texttt{STMTS}^b \; \}
\end{array}
$$

Assumptions:

- interface names in B are distinct;
- field names in $\texttt{FDS}^b$ are distinct;
- method (constructor) name/parameters (parameters) in $\texttt{MDS}^b$ ($\texttt{KDS}^b$) are distinct;
- class names in $\texttt{CATCHES}^b$ are distinct.

Figure 4.2: Syntax - Binaries

- *Field accesses/assignments*. Each field access/assignment is annotated with the static type of the expression on which the field is accessed/assigned, and the type T of the declared type of the field. As instance field accesses and static field accesses are translated into different JVM instructions (respectively getfield and getstatic), we use two different kinds of annotation: $\ll$ C.T $\gg_{\texttt{if}}$ for instance fields and $\ll$ RT.T $\gg_{\texttt{sf}}$ for static fields. In the former case the annotation consists of the class name (interfaces cannot declare instance fields) and the type of the field T; in the latter case the annotation consists of the reference type (either a class or an interface name) and the type of the field T.

- *Method invocations*. We model four different kinds of method invocations; each one corresponds to a different JVM instruction and needs a different kind of annotation. The kinds are:

  – *Virtual*, used for instance method invocations. The annotation $\ll$ C.T ($\bar{\texttt{T}}$) $\gg_{\texttt{vrt}}$ consists of the type of the receiver C (a class because invocations via an interface require a special, different, bytecode to be emitted), the return type T and the parameter types $\bar{\texttt{T}}$. For simplicity, we do not distinguish between invocations of private and non-private methods, even though an invocation of a private method in Java uses a particular JVM instruction different from the "normal" instance method invocation. The reason is that a private method cannot be overridden, so the treatment is analogous to the static case (JLS 15.12.3).

  – *Super*, used for invoking a method defined in the superclass. The annotation $\ll$ C.T ($\bar{\texttt{T}}$) $\gg_{\texttt{spr}}$ consists of the name of the superclass C, the return type T and the parameter types $\bar{\texttt{T}}$. Note that there is no corresponding super-annotation for field accesses, because a field access has a static binding and there is no difference in using super.f or C.f (assuming C to be the name of superclass). As for constructor invocation annotations, the annotation contains the (redundant) name of the superclass for mimicking the JVM instruction invokespecial which is used to compile non-virtual invocations.

  – *Static*, used for static method invocations. The annotation $\ll$ C.T ($\bar{\texttt{T}}$) $\gg_{\texttt{stt}}$ consists of the type of the receiver C (a class because interfaces cannot declare static methods), the return type T and the parameter types $\bar{\texttt{T}}$.

  – *Interface*, used for invocations where the static type of the receiver is an interface I. In this case, the annotation $\ll$ I.T ($\bar{\texttt{T}}$) $\gg_{\texttt{int}}$ consists of the name of the interface I, the return type T and the type of the parameters $\bar{\texttt{T}}$.

$$
\begin{array}{lcl}
\Gamma^{\mathrm{s}} & ::= & \gamma^{\mathrm{s}}_1 \ldots \gamma^{\mathrm{s}}_n \\
\gamma^{\mathrm{s}} & ::= & \mathtt{C} \mapsto [\textsc{type}{=}\mathtt{class}, \textsc{am}{=}\mathtt{AM}, \textsc{ck}{=}\mathtt{CK}, \textsc{parent}{=}\mathtt{C}', \textsc{is}{=}\mathtt{I}_1 \ldots \mathtt{I}_n, \textsc{kss}{=}\mathtt{KSS}, \textsc{fss}{=}\mathtt{FSS}, \textsc{nhs}{=}\mathtt{NHS}] \mid \\
 & & \mathtt{I} \mapsto [\textsc{type}{=}\mathtt{interface}, \textsc{am}{=}\mathtt{AM}, \textsc{is}{=}\mathtt{I}_1 \ldots \mathtt{I}_n, \textsc{fss}{=}\mathtt{FSS}, \textsc{nhs}{=}\mathtt{NHS}]
\end{array}
$$

$$
\begin{array}{lcl}
\bar{\mathtt{T}} & ::= & \mathtt{T}_1 \ldots \mathtt{T}_n \\
\mathtt{T}^{\perp} & ::= & \mathtt{T} \mid \perp \\
\bar{\mathtt{T}}^{\perp} & ::= & \mathtt{T}^{\perp}_1 \ldots \mathtt{T}^{\perp}_n \\
\mathtt{KS} & ::= & \mathtt{AM}\ \bar{\mathtt{T}}\ \mathtt{throws\ ES} \\
\mathtt{KSS} & ::= & \mathtt{KS}_1 \ldots \mathtt{KS}_n \\
\mathtt{FS} & ::= & \mathtt{AM\ FINAL\ FK\ T\ f} \mid \mathtt{AM\ final\ static\ T\ f} \ = \ \mathtt{E}^s \\
\mathtt{FSS} & ::= & \mathtt{FS}_1 \ldots \mathtt{FS}_n \\
\mathtt{NH} & ::= & \mathtt{AM\ MK\ T\ m}(\bar{\mathtt{T}})\ \mathtt{throws\ ES} \\
\mathtt{NHS} & ::= & \mathtt{NH}_1 \ldots \mathtt{NH}_n
\end{array}
$$

Figure 4.3: Type environments

4.2.2 Type environments

In this chapter, since we focus on selective recompilation, we use standard type environments only. A (standard) type environment Γ^{s}, is a sequence of type assignments γ^{s}, which maps class/interface names to their respective types. Type environments are defined in Figure 4.3.

The assignment $\mathtt{C} \mapsto [\textsc{type}{=}\mathtt{class}, \textsc{am}{=}\mathtt{AM}, \textsc{ck}{=}\mathtt{CK}, \textsc{parent}{=}\mathtt{C}', \textsc{is}{=}\mathtt{I}_1 \ldots \mathtt{I}_n, \textsc{kss}{=}\mathtt{KSS}, \textsc{fss}{=}\mathtt{FSS}, \textsc{nhs}{=}\mathtt{NHS}]$ has the meaning "the class `C` has access modifier `AM` and kind `CK`, extends `C`', implements $\mathtt{I}_1 \ldots \mathtt{I}_n$ and has constructor signatures `KSS`, field signatures `FSS` and naked (method) headers `NHS`".

A naked (method) header is a method header without the parameter names; roughly speaking it is the "signature" of the method, but this terminology would conflict with the one used by [GJSB00], where a method signature is just the pair consisting of the method name and parameter types. In the following we use the terms "naked (method) header" and "method header" interchangeably where there is no ambiguity.

Analogously, a type assignment $\mathtt{I} \mapsto [\textsc{type}{=}\mathtt{interface}, \textsc{am}{=}\mathtt{AM}, \textsc{is}{=}\mathtt{I}_1 \ldots \mathtt{I}_n, \textsc{fss}{=}\mathtt{FSS}, \textsc{nhs}{=}\mathtt{NHS}]$ has the meaning "the interface `I` has access modifier `AM`, extends $\mathtt{I}_1 \ldots \mathtt{I}_n$, and has field signatures `FSS` and method headers `NHS`".

In the following we use the dot notation to extract a component from a class/interface type. For instance, we use $\Gamma^{\mathrm{s}}(\mathtt{C}).\textsc{parent} = \mathtt{C}'$ as a shortcut for
$\Gamma^{\mathrm{s}}(\mathtt{C}) \mapsto [\textsc{type}{=}\mathtt{class}, \textsc{am}{=}\_, \textsc{ck}{=}\_, \textsc{parent}{=}\mathtt{C}', \textsc{is}{=}\_, \textsc{kss}{=}\_, \textsc{fss}{=}\_, \textsc{nhs}{=}\_]$.

In order to handle ctc-fields (see Section 4.1.1), a field signature `FS` of a `final static` field may contain, among the other information, the *initialization expression* of the field. Any final static field with an initialization expression $\mathtt{E}^s$ is a ctc-field when $\mathtt{E}^s$ can be evaluated, at compile-time, to a value of primitive type.

$\mathcal{T}(\texttt{AM CK class C extends C}' \texttt{ implements I}_1 \ldots \texttt{I}_n \ \{ \ \texttt{KDS}^s \ \texttt{FDS}^s \ \texttt{MDS}^s \ \}) =$
$\quad \texttt{C} \mapsto [\textsc{type}{=}\texttt{class}, \textsc{am}{=}\texttt{AM}, \textsc{ck}{=}\texttt{CK}, \textsc{parent}{=}\texttt{C}', \textsc{is}{=}\texttt{I}_1 \ldots \texttt{I}_n, \textsc{kss}{=}\mathcal{T}(\texttt{KDS}^s), \textsc{fss}{=}\mathcal{T}(\texttt{FDS}^s), \textsc{nhs}{=}\mathcal{T}(\texttt{MDS}^s)]$
$\mathcal{T}(\texttt{AM CK class C extends C}' \texttt{ implements I}_1 \ldots \texttt{I}_n \ \{ \ \texttt{KDS}^b \ \texttt{FDS}^b \ \texttt{MDS}^b \ \}) =$
$\quad \texttt{C} \mapsto [\textsc{type}{=}\texttt{class}, \textsc{am}{=}\texttt{AM}, \textsc{ck}{=}\texttt{CK}, \textsc{parent}{=}\texttt{C}', \textsc{is}{=}\texttt{I}_1 \ldots \texttt{I}_n, \textsc{kss}{=}\mathcal{T}(\texttt{KDS}^b), \textsc{fss}{=}\mathcal{T}(\texttt{FDS}^b), \textsc{nhs}{=}\mathcal{T}(\texttt{MDS}^b)]$

$\mathcal{T}(\texttt{AM interface I extends I}_1 \ldots \texttt{I}_n \ \{ \ \texttt{FDS}^s \ \texttt{MDS}^s \ \}) =$
$\qquad \texttt{I} \mapsto [\textsc{type}{=}\texttt{interface}, \textsc{am}{=}\texttt{AM}, \textsc{is}{=}\texttt{I}_1 \ldots \texttt{I}_n, \textsc{fss}{=}\mathcal{T}(\texttt{FDS}^s), \textsc{nhs}{=}\mathcal{T}(\texttt{MDS}^s)]$
$\mathcal{T}(\texttt{AM interface I extends I}_1 \ldots \texttt{I}_n \ \{ \ \texttt{FDS}^b \ \texttt{MDS}^b \ \}) =$
$\qquad \texttt{I} \mapsto [\textsc{type}{=}\texttt{interface}, \textsc{am}{=}\texttt{AM}, \textsc{is}{=}\texttt{I}_1 \ldots \texttt{I}_n, \textsc{fss}{=}\mathcal{T}(\texttt{FDS}^b), \textsc{nhs}{=}\mathcal{T}(\texttt{MDS}^b)]$

$\mathcal{T}(\texttt{KD}_1^s \ldots \ \texttt{KD}_n^s) = \mathcal{T}(\texttt{KD}_1^s) \ldots \mathcal{T}(\texttt{KD}_n^s)$
$\mathcal{T}(\texttt{KD}_1^b \ldots \ \texttt{KD}_n^b) = \mathcal{T}(\texttt{KD}_1^b) \ldots \mathcal{T}(\texttt{KD}_n^b)$
$\mathcal{T}(\texttt{FD}_1^s \ldots \ \texttt{FD}_n^s) = \mathcal{T}(\texttt{FD}_1^s) \ldots \mathcal{T}(\texttt{FD}_n^s)$
$\mathcal{T}(\texttt{FD}_1^b \ldots \ \texttt{FD}_n^b) = \mathcal{T}(\texttt{FD}_1^b) \ldots \mathcal{T}(\texttt{FD}_n^b)$
$\mathcal{T}(\texttt{MD}_1^s \ldots \ \texttt{MD}_n^s) = \mathcal{T}(\texttt{MD}_1^s) \ldots \mathcal{T}(\texttt{MD}_n^s)$
$\mathcal{T}(\texttt{MD}_1^b \ldots \ \texttt{MD}_n^b) = \mathcal{T}(\texttt{MD}_1^b) \ldots \mathcal{T}(\texttt{MD}_n^b)$

$\mathcal{T}(\texttt{AM} \ (\texttt{T}_1 \ \texttt{x}_1, \ldots, \texttt{T}_n \ \texttt{x}_n) \ \texttt{throws ES} \ \{ \ \ldots \ \}) = \texttt{AM T}_1 \ldots \texttt{T}_n \ \texttt{throws ES}$

$\mathcal{T}(\texttt{AM FINAL FK T f} = \texttt{E}^s \ ;) = \texttt{AM FINAL FK T f} \qquad$ if $\texttt{FINAL} = \epsilon$ or $\texttt{FK} = \epsilon$
$\mathcal{T}(\texttt{AM final static T f} = \texttt{E}^s \ ;) = \texttt{AM final static T f} \ = \ \texttt{E}^s$
$\mathcal{T}(\texttt{AM FINAL FK T f} = \texttt{E}^b \ ;) = \texttt{AM FINAL FK T f} \qquad$ if $\texttt{FINAL} = \epsilon$ or $\texttt{FK} = \epsilon$
$\mathcal{T}(\texttt{AM final static T f} = \texttt{E}^b \ ;) = \texttt{AM final static T f} \ = \ \nu \qquad$ if $\texttt{E}^b$ is the value ν.
$\mathcal{T}(\texttt{AM final static T f} = \texttt{E}^b \ ;) = \texttt{AM final static T f} \qquad$ if $\texttt{E}^b$ is not a value.

$\mathcal{T}(\texttt{AM abstract T m}(\texttt{T}_1 \ \texttt{x}_1, \ldots, \texttt{T}_n \ \texttt{x}_n) \ \texttt{throws ES} \ ;) = \texttt{AM abstract T m}(\texttt{T}_1 \ldots \texttt{T}_n) \ \texttt{throws ES}$
$\mathcal{T}(\texttt{AM MK T m}(\texttt{T}_1 \ \texttt{x}_1, \ldots, \texttt{T}_n \ \texttt{x}_n) \ \texttt{throws ES} \ \{ \ \ldots \ \}) = \texttt{AM MK T m}(\texttt{T}_1 \ldots \texttt{T}_n) \ \texttt{throws ES}$

Figure 4.4: Definition of the type extraction function $\mathcal{T}$.

A type environment can be extracted from source and binary fragments using the function $\mathcal{T}$ defined in Figure 4.4. This function discards all code retaining just type information; the only exception are final static fields with an initialization expression which can be ctc-fields. In this case, the initialization expressions must be retained as well. When extracting type information from source fragments, a (trivial) optimization can be performed: if an expression $\mathtt{E}^s$ contains a method invocation or a `new` expression, then $\mathtt{E}^s$ can be never evaluated at compile-time. Hence, a field initialized by $\mathtt{E}^s$ can never be a ctc-field and its initialization expression $\mathtt{E}^s$ can be ignored. This reasoning does not apply to binary fragments because the initialization expression of a ctc-field is, by definition, compiled directly into a value, so any (`final static`) field which is initialized by anything but a value is not a ctc-field.

Because compile-time constant initialization expressions, contained in the sources, are lost in the translated binaries, the type information for a fragment must be always extracted from its source. This is not a real limitation: if only the binary is available for, say, class `C`, then `C` is part of an external library and:

- the values of ctc-fields declared by `C` are *really* constant (that is, they cannot change because of a recompilation) being `C` available only in binary form;
- it is safe to assume that, being part of an external library, class `C` does not depend on our sources (as a result, it never has to be recompiled because our sources has changed).

Our selective recompilation strategy requires the extraction of a type environment Γ^{S} from the fragments each time the selective recompilation is run. So, the approach of extracting type information from sources, which requires to parse them, may seem, at first sight, very expensive. Fortunately, some simple considerations show it is indeed feasible:

- the type information extracted for each source can be cached;
- when a source is changed, it must be recompiled, therefore it must be parsed anyway (and, assuming the compilation manager to be tightly integrated with the compiler, the parse tree used to extract type information can be directly used by the compiler at a later stage).

4.2.3 Type assumptions

Type assumptions γ, defined in Figure 4.5, describe fine-grained type requirements. Note that, differently from the assumptions of the type system given in the previous chapter,

$$
\begin{array}{rcl}
\gamma & ::= & \mathtt{T} \leq \mathtt{T}' \;\mid \\
 & & \mathtt{RT} \not\prec \exists\, \mathtt{T} \;\mid \\
 & & \mathtt{RT} \not\prec \exists_{\mathtt{cls}}\, \mathtt{CK}^\star\ \mathtt{C} \mid \\
 & & \mathtt{RT} \not\prec \exists_{\mathtt{ifc}}\, \mathtt{I} \;\mid \\
 & & \mathtt{RT} \not\prec \mathit{Cns}(\mathtt{C}, \bar{\mathtt{T}}^\perp) = [\textsc{par}{=}\bar{\mathtt{T}}] \;\mid \\
 & & \mathtt{RT} \not\prec \mathit{Fld}(\mathtt{RT}', \mathtt{f}) = [\textsc{final}{=}\mathtt{FINAL}^\star, \textsc{fk}{=}\mathtt{FK}, \textsc{t}{=}\mathtt{T}] \;\mid \\
 & & \mathtt{RT} \not\prec \mathit{Mth}(\mathtt{RT}', \mathtt{m}, \bar{\mathtt{T}}^\perp) = [\textsc{mk}{=}\mathtt{MK}^\star, \textsc{ret}{=}\mathtt{T}, \textsc{par}{=}\bar{\mathtt{T}}] \mid \\
 & & \mathtt{RT} \not\prec \mathtt{E}^s = \nu \mid \\
 & & \mathtt{RT} \not\prec \mathtt{EE} \supseteq_{\mathtt{Exc}} \mathtt{ES} \\
 & & \mathtt{RT} \not\prec \mathtt{EE} \subseteq_{\mathtt{Exc}} \mathtt{ES} \\
 & & \\
\mathtt{EE} & ::= & \varepsilon(\mathtt{RT} \not\prec \mathtt{C}(\bar{\mathtt{T}})) \mid \varepsilon(\mathtt{RT} \not\prec \mathtt{RT}'.\mathtt{m}(\bar{\mathtt{T}}^\perp)) \mid \mathtt{ES} \mid \mathtt{EE}_1 \cup_{\mathtt{Exc}} \mathtt{EE}_2 \mid \mathtt{EE} \setminus_{\mathtt{Exc}} \mathtt{ES} \\
 & & \\
\mathtt{CK}^\star & ::= & \mathtt{CK} \mid \mathtt{any} \\
\mathtt{FINAL}^\star & ::= & \mathtt{FINAL} \mid \mathtt{any} \\
\mathtt{MK}^\star & ::= & \mathtt{MK} \mid \mathtt{not\text{-}static}
\end{array}
$$

Figure 4.5: Type assumptions

these assumptions describe requirements which directly affect code generation only. Well-formedness of type environments is dealt separately, as detailed in Section 4.2.6.

In order to typecheck expressions we need to introduce the type of `null`, called $\perp$. A type $\mathtt{T}^\perp$, defined in Figure 4.3, can be either a regular type `T` (which can be used in source programs) or $\perp$ (which, conversely, cannot be used in source programs). The assumptions are:

- $\mathtt{T} \leq \mathtt{T}'$ with the meaning "`T` is a subtype of $\mathtt{T}'$";
- $\mathtt{RT} \not\prec \exists\, \mathtt{T}$ with the meaning "type `T` exists and is accessible from code contained in `RT`"[8];
- $\mathtt{RT} \not\prec \exists_{\mathtt{cls}}\, \mathtt{CK}^\star\ \mathtt{C}$ with the meaning "class `C`, with kind $\mathtt{CK}^\star$, exists and is accessible from code contained in `RT`". $\mathtt{CK}^\star = \mathtt{any}$ means "any kind", that is, we do not care whether the class is abstract or not;
- $\mathtt{RT} \not\prec \exists_{\mathtt{ifc}}\, \mathtt{I}$ with the meaning "interface `I` exists and is accessible from code contained in `RT`";

[8] If the reader thinks the symbol "$\not\prec$" as an eye, then she/he can interpret any assumption of the form "$\mathtt{RT} \not\prec \ldots$" as: "`RT` *sees* ..." and this is supposed to help ☺.

- $\mathtt{RT} \not\lessdot \mathit{Cns}(\mathtt{C}, \bar{\mathtt{T}}^{\perp}) = [\textsc{par}{=}\bar{\mathtt{T}}]$ with the meaning "the most specific constructor for class C and argument types $\bar{\mathtt{T}}^{\perp}$, invoked from code contained in RT, has parameter types $\bar{\mathtt{T}}$;

- $\mathtt{RT} \not\lessdot \mathit{Fld}(\mathtt{RT}', \mathtt{f}) = [\textsc{final}{=}\mathtt{FINAL}^{\star}, \textsc{fk}{=}\mathtt{FK}, \textsc{t}{=}\mathtt{T}]$ with the meaning "if code contained in RT looks up a field named f in type RT′, then it finds a field with a final modifier $\mathtt{FINAL}^{\star}$, kind FK and type T". $\mathtt{FINAL}^{\star} = \mathtt{any}$ means we do not care whether the field is final or not;

- $\mathtt{RT} \not\lessdot \mathit{Mth}(\mathtt{RT}', \mathtt{m}, \bar{\mathtt{T}}^{\perp}) = [\textsc{mk}{=}\mathtt{MK}^{\star}, \textsc{ret}{=}\mathtt{T}, \textsc{par}{=}\bar{\mathtt{T}}]$ with the meaning "the most specific method, invoked from code contained in RT, for a method named m, with argument types $\bar{\mathtt{T}}^{\perp}$ on a receiver with static type RT′ is a method which has kind $\mathtt{MK}^{\star}$, return type T and parameter types $\bar{\mathtt{T}}$";

- $\mathtt{RT} \not\lessdot \mathtt{E}^{s} = \nu$ with the meaning "inside RT, the expression $\mathtt{E}^{s}$ is a compile-time constant with value ν";

- $\mathtt{RT} \not\lessdot \mathtt{EE} \supseteq_{\mathtt{Exc}} \mathtt{ES}$ and $\mathtt{RT} \not\lessdot \mathtt{EE} \subseteq_{\mathtt{Exc}} \mathtt{ES}$ with the meaning: "inside RT the exception expression EE contains (resp., is contained in) the exception set ES".

Exception expressions EE describe a set of exceptions in an abstract way. The exception expression $\varepsilon(\mathtt{RT} \not\lessdot \mathtt{C}(\bar{\mathtt{T}}))$ represents the exception specification for the constructor of class C with argument type $\bar{\mathtt{T}}$.

The exception expression $\varepsilon(\mathtt{RT} \not\lessdot \mathtt{RT}'.\mathtt{m}(\bar{\mathtt{T}}^{\perp}))$ represents the exception specification of *the most specific method* named m, for an invocation with argument types $\bar{\mathtt{T}}^{\perp}$, on an object of type RT'. Note the asymmetry: in the former case the constructor is fixed, while in the latter case the method is defined as "the most specific for…". This difference is due in how constructor invocations (via super or new) and method invocations are annotated. When a constructor invocation, on a class C, is annotated with $\bar{\mathtt{T}}$, then the selected constructor is always the *only* constructor with parameter types $\bar{\mathtt{T}}$ declared in class C (constructors are never inherited). On the other hand, if a method invocation is annotated with $\mathtt{RT.T}(\bar{\mathtt{T}})$, then the most specific method can be declared in RT or in *any* of its supertypes, because the type RT, contained in the annotation, specifies the static type of the receiver, not the declaring type (since Java 1.4; in previous versions the reference type in method annotation was the declaring type, so there would be no difference between constructors and methods if we modeled an earlier version of Java).

These exception expressions, and also the actual sets of exceptions ES, can be combined together using the operators "union" $\cup_{\mathtt{Exc}}$ and "minus" $\setminus_{\mathtt{Exc}}$ which, in first approximation, can be thought as the corresponding set operations. However, they have to deal with the fact that any exception C actually stands for "C or any subclass"; see Figure 4.14 for more details.

The rules defining the judgment $\Gamma^{\mathrm{S}} \vdash \gamma$ are shown in Figures from 4.6 to 4.14. For the time being we assume Γ^{S} to be a well-formed environment; that is, we assume its type hierarchy is acyclic and the Java rules on method overriding/hiding are respected. In Section 4.2.6 we formalize this idea and discuss how we can incrementally check it.

Figure 4.6 shows the metarules for deriving subtyping judgments. The first five metarules define subtyping between types, while the rules (6)-(8) define subtyping between access modifiers and method kinds. These judgments are then used to define a subtyping between method headers and between field signatures which are used when typechecking class and interface declarations.

The primitive types, `int` and `bool`, are subtypes of themselves only (metarule (1)). A class `C` is a subtype of itself, its (direct) superclass, and every implemented interface $\mathtt{I}_j$ (metarule (2)). Analogously, an interface `I` is a subtype of itself, `Object`, and every extended interface $\mathtt{I}_j$ (metarule (3)).

The type of `null`, $\perp$, is a subtype of any reference type, that is, every type except for primitive ones (last consequence in both metarules (2) and (3)).

Subtyping relation is transitively closed and extended to sequence of types in the usual way (metarules (4) and (5)).

The subtyping between access modifiers, defined in metarules (6) and (7), and between method kinds, defined in metarule (8), are used to enforce the requirements in method overriding (JLS 8.4.6.3). That is:

1. the access modifier of an overriding (or hiding for the `static` case) method must provide at least as much access as the overridden (or hidden) method (JLS 8.4.6.3);
2. an instance method cannot override a static method and vice versa (JLS 8.4.6.1 and JLS 8.4.6.2).

A method header $\mathtt{AM}_1\ \mathtt{MK}_1\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})\ \mathtt{throws}\ \mathtt{ES}_1$ is a subtype of another method header $\mathtt{AM}_2\ \mathtt{MK}_2\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})\ \mathtt{throws}\ \mathtt{ES}_2$ when:

- they have the same name `m`, the same parameter types $\bar{\mathtt{T}}$ and the same return type `T`;
- the access modifier of the former provides at least as much access as the latter, $\Gamma^{\mathrm{S}} \vdash \mathtt{AM}_1 \leq \mathtt{AM}_2$;
- they are both instance (ϵ or `abstract`) or both `static`, $\Gamma^{\mathrm{S}} \vdash \mathtt{MK}_1 \leq \mathtt{MK}_2$;
- the former may only throws exceptions which are subclasses of the ones thrown by the latter, $\Gamma^{\mathrm{S}} \vdash \mathtt{ES}_2 \supseteq_{\mathtt{Exc}} \mathtt{ES}_1$.

$$(1)\ \frac{}{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \texttt{int} \leq \texttt{int} \\ \Gamma^{\mathrm{S}} \vdash \texttt{bool} \leq \texttt{bool}\end{array}}$$

$$(2)\ \frac{\begin{array}{l}\Gamma^{\mathrm{S}}(\texttt{C}).\textsc{parent} = \texttt{C}' \\ \Gamma^{\mathrm{S}}(\texttt{C}).\textsc{is} = \texttt{I}_1 \ldots \texttt{I}_n\end{array}}{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \texttt{C} \leq \texttt{C} \\ \Gamma^{\mathrm{S}} \vdash \texttt{C} \leq \texttt{C}' \\ \Gamma^{\mathrm{S}} \vdash \texttt{C} \leq \texttt{I}_j \\ \Gamma^{\mathrm{S}} \vdash \bot \leq \texttt{C}\end{array}} \qquad (3)\ \frac{\Gamma^{\mathrm{S}}(\texttt{I}).\textsc{is} = \texttt{I}_1 \ldots \texttt{I}_n}{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \texttt{I} \leq \texttt{I} \\ \Gamma^{\mathrm{S}} \vdash \texttt{I} \leq \texttt{Object} \\ \Gamma^{\mathrm{S}} \vdash \texttt{I} \leq \texttt{I}_j \\ \Gamma^{\mathrm{S}} \vdash \bot \leq \texttt{I}\end{array}}$$

$$(4)\ \frac{\Gamma^{\mathrm{S}} \vdash \texttt{RT}_1 \leq \texttt{RT}_2 \quad \Gamma^{\mathrm{S}} \vdash \texttt{RT}_2 \leq \texttt{RT}_3}{\Gamma^{\mathrm{S}} \vdash \texttt{RT}_1 \leq \texttt{RT}_3} \qquad (5)\ \frac{\Gamma^{\mathrm{S}} \vdash \texttt{T}_1 \leq \texttt{T}'_1 \ \ldots \ \texttt{T}_n \leq \texttt{T}'_n}{\Gamma^{\mathrm{S}} \vdash \texttt{T}_1 \ldots \texttt{T}_n \leq \texttt{T}'_1 \ldots \texttt{T}'_n}$$

$$(6)\ \frac{}{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \texttt{public} \leq \texttt{protected} \\ \Gamma^{\mathrm{S}} \vdash \texttt{protected} \leq \epsilon \\ \Gamma^{\mathrm{S}} \vdash \epsilon \leq \texttt{private} \\ \Gamma^{\mathrm{S}} \vdash \texttt{AM} \leq \texttt{AM}\end{array}} \qquad (7)\ \frac{\Gamma^{\mathrm{S}} \vdash \texttt{AM}_1 \leq \texttt{AM}_2 \quad \Gamma^{\mathrm{S}} \vdash \texttt{AM}_2 \leq \texttt{AM}_3}{\Gamma^{\mathrm{S}} \vdash \texttt{AM}_1 \leq \texttt{AM}_3}$$

$$(8)\ \frac{}{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \texttt{MK} \leq \texttt{MK} \\ \Gamma^{\mathrm{S}} \vdash \texttt{abstract} \leq \epsilon \\ \Gamma^{\mathrm{S}} \vdash \epsilon \leq \texttt{abstract}\end{array}}$$

$$(9)\ \frac{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \texttt{AM}_1 \leq \texttt{AM}_2 \\ \Gamma^{\mathrm{S}} \vdash \texttt{MK}_1 \leq \texttt{MK}_2 \\ \Gamma^{\mathrm{S}} \vdash \texttt{ES}_2 \supseteq_{\texttt{Exc}} \texttt{ES}_1\end{array}}{\Gamma^{\mathrm{S}} \vdash \texttt{AM}_1\ \texttt{MK}_1\ \texttt{T}\ \texttt{m}(\bar{\texttt{T}})\ \texttt{throws}\ \texttt{ES}_1 \leq \texttt{AM}_2\ \texttt{MK}_2\ \texttt{T}\ \texttt{m}(\bar{\texttt{T}})\ \texttt{throws}\ \texttt{ES}_2}$$

$$(10)\ \frac{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \texttt{RT}_1 \leq \texttt{RT}_2 \\ \Gamma^{\mathrm{S}} \vdash \bar{\texttt{T}}_1 \leq \bar{\texttt{T}}_2\end{array}}{\Gamma^{\mathrm{S}} \vdash \texttt{RT}_1.\_\ \_\ \_\ \texttt{m}(\bar{\texttt{T}}_1)\ \texttt{throws}\ \_ \leq \texttt{RT}_2.\_\ \_\ \_\ \texttt{m}(\bar{\texttt{T}}_2)\ \texttt{throws}\ \_}$$

$$(11)\ \frac{\Gamma^{\mathrm{S}} \vdash \texttt{RT}_1 \leq \texttt{RT}_2}{\Gamma^{\mathrm{S}} \vdash \texttt{RT}_1.\texttt{FS}_1 \leq \texttt{RT}_2.\texttt{FS}_2} \qquad \begin{array}{l}\texttt{FS}_1 = \texttt{AM}_1\ \texttt{FK}_1\ \texttt{T}_1\ \texttt{f} \ldots \\ \texttt{FS}_2 = \texttt{AM}_2\ \texttt{FK}_2\ \texttt{T}_2\ \texttt{f} \ldots\end{array}$$

Figure 4.6: Rules defining subtyping

Annotated method headers, which we will use when resolving overloading, consist of method headers prefixed by the name of the declaring types (of the methods). We extend subtyping of method headers to annotated method headers in metarule (10): an annotated method header $\mathtt{RT}_1.\mathtt{NH}_1$ is a subtype of another $\mathtt{RT}_2.\mathtt{NH}_2$ when both the declaring type and parameter types of the former are subtypes of the declaring type and the parameter types of the latter. This definition corresponds to the definition of more specific methods given in JLS 15.12.2.2.

Finally, in metarule (11), we define an annotated field signature (which is the analogous of an annotated method header) as a subtype of another when they declare a field with the same name `f` and the declaring type of the former is a subtype of the declaring type of the latter. This corresponds to the fact that, in contrast to what happens for methods, there are no requirements when a field hides inherited fields (JLS 8.3).

Figure 4.7 shows the metarules for deriving judgments related to accessibility. Primitive types can be accessed by any type (metarule (1)). Classes and interfaces can be accessed by any other type when they are `public` (metarules (2) and (5)), but only by members in their own package when they have a default access modifier ϵ – JLS 6.6.1 (metarules (3) and (6)). As we do not model `import` keyword, we assume that every name `C` and `I` is fully-qualified, so the declaring package of any name can be extracted by the function *pck*. Metarules (4) and (7) define that any accessible class or interface can be seen as an accessible type.

The auxiliary predicate $isAcc(\Gamma^{\mathrm{s}}, \mathtt{RT}, \mathtt{RT}', \mathtt{AM})$, defined at the bottom of Figure 4.7, is true when, in a type environment Γ^{s}, the code contained in a type `RT` can access a member of $\mathtt{RT}'$ declared with access modifier `AM` as described in JLS 6.6.1.

Figure 4.8 contains the definitions of some auxiliary functions on sets of annotated method headers `ANHS`:

1. $nonPrivate(\mathtt{ANHS})$ returns all the non-private members contained in `ANHS`;
2. $acc_{\Gamma^{\mathrm{s}}}(\mathtt{RT}, \mathtt{KSS})$ returns all the constructor signatures `KS` contained in `KSS` which are accessible from `RT`;
3. $acc_{\Gamma^{\mathrm{s}}}(\mathtt{RT}, \mathtt{AFSS})$ returns all the annotated field signatures `AFS` contained in `AFSS` which are accessible from `RT`;
4. $accNP_{\Gamma^{\mathrm{s}}}(\mathtt{RT}, \mathtt{AFSS})$ returns all the non-private annotated field signatures `AFS` contained in `AFSS` which are accessible from `RT` (note that being accessible does not imply to be non-private, see the explanation below);
5. $acc_{\Gamma^{\mathrm{s}}}(\mathtt{RT}, \mathtt{ANHS})$ returns all the annotated method headers `ANH` contained in `ANHS` which are accessible from `RT`;

$$\text{(1)}\ \frac{}{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists\,\mathtt{int} \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists\,\mathtt{bool}\end{array}}$$

$$\text{(2)}\ \frac{\Gamma^{\mathrm{S}}(\mathtt{C}).\textsc{type} = \mathtt{class} \quad \Gamma^{\mathrm{S}}(\mathtt{C}).\textsc{am} = \mathtt{public} \quad \Gamma^{\mathrm{S}}(\mathtt{C}).\textsc{ck} = \mathtt{CK}}{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists_{\mathtt{cls}}\,\mathtt{CK}\ \mathtt{C}}$$

$$\text{(3)}\ \frac{\Gamma^{\mathrm{S}}(\mathtt{C}).\textsc{type} = \mathtt{class} \quad \Gamma^{\mathrm{S}}(\mathtt{C}).\textsc{am} = \epsilon \quad \Gamma^{\mathrm{S}}(\mathtt{C}).\textsc{ck} = \mathtt{CK}}{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists_{\mathtt{cls}}\,\mathtt{CK}\ \mathtt{C}} \qquad pck(\mathtt{RT}) = pck(\mathtt{C})$$

$$\text{(4)}\ \frac{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists_{\mathtt{cls}}\,\_\ \mathtt{C}}{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists_{\mathtt{cls}}\,\mathtt{any}\ \mathtt{C} \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists\,\mathtt{C}\end{array}}$$

$$\text{(5)}\ \frac{\Gamma^{\mathrm{S}}(\mathtt{I}).\textsc{type} = \mathtt{interface} \quad \Gamma^{\mathrm{S}}(\mathtt{I}).\textsc{am} = \mathtt{public}}{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists_{\mathtt{ifc}}\,\mathtt{I}}$$

$$\text{(6)}\ \frac{\Gamma^{\mathrm{S}}(\mathtt{I}).\textsc{type} = \mathtt{interface} \quad \Gamma^{\mathrm{S}}(\mathtt{I}).\textsc{am} = \epsilon}{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists_{\mathtt{ifc}}\,\mathtt{I}} \qquad pck(\mathtt{RT}) = pck(\mathtt{I})$$

$$\text{(7)}\ \frac{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists_{\mathtt{ifc}}\,\mathtt{I}}{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists\,\mathtt{I}}$$

$$isAcc(\Gamma^{\mathrm{S}}, \mathtt{RT}, \mathtt{RT}', \mathtt{AM}) = \begin{cases} \mathtt{RT} = \mathtt{RT}' & \text{if } \mathtt{AM} = \mathtt{private} \\ pck(\mathtt{RT}) = pck(\mathtt{RT}') & \text{if } \mathtt{AM} = \epsilon \\ pck(\mathtt{RT}) = pck(\mathtt{RT}') \vee \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \leq \mathtt{RT}' & \text{if } \mathtt{AM} = \mathtt{protected} \\ \mathtt{true} & \text{if } \mathtt{AM} = \mathtt{public} \end{cases}$$

Figure 4.7: Accessibility (1/2)

$$
\begin{array}{l}
nonPrivate(\texttt{ANHS}) = \{\texttt{RT.AM}\ldots \in \texttt{AFSS}|\texttt{AM} \neq \texttt{private}\} \\
\\
acc_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{KSS}) = \{\texttt{KS} \in \texttt{KSS}|\texttt{KS} = \texttt{AM}\ldots, isAcc(\Gamma^{\mathrm{S}},\texttt{RT},\texttt{RT}',\texttt{AM})\,\} \\
\\
acc_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{AFSS}) = \{\texttt{AFS} \in \texttt{AFSS}|\texttt{AFS} = \texttt{RT}'.\texttt{AM}\ldots, isAcc(\Gamma^{\mathrm{S}},\texttt{RT},\texttt{RT}',\texttt{AM})\,\} \\
accNP_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{AFSS}) = nonPrivate(\texttt{AFSS}) \cap acc_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{AFSS}) \\
\\
acc_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS}) = \{\texttt{ANH} \in \texttt{ANHS}|\texttt{ANH} = \texttt{RT}'.\texttt{AM}\ldots, isAcc(\Gamma^{\mathrm{S}},\texttt{RT},\texttt{RT}',\texttt{AM})\,\} \\
accNP_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS}) = nonPrivate(\texttt{ANHS}) \cap acc_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS}) \\
nonAcc_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS}) = \texttt{ANHS} \setminus accNP_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS}) \\
nonAccNP_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS}) = nonPrivate(\texttt{ANHS}) \setminus accNP_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS}) \\
\\
abs(\texttt{ANHS}) = \{\texttt{RT.AM abstract}\ldots \in \texttt{ANHS}\} \\
nonAbs(\texttt{ANHS}) = \texttt{ANHS} \setminus abs(\texttt{ANHS}) \\
ann(\texttt{RT},\texttt{NHS}) = \{\texttt{RT.NH}|\texttt{NH} \in \texttt{NHS}\}
\end{array}
$$

Figure 4.8: Accessibility (2/2)

6. $accNP_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS})$ returns all the non-private annotated method headers `ANH` contained in `ANHS` which are accessible from `RT`;

7. $nonAcc_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS})$ returns all the annotated method headers `ANH` contained in `ANHS` which are not accessible from `RT`;

8. $nonAccNP_{\Gamma^{\mathrm{S}}}(\texttt{RT},\texttt{ANHS})$ returns all the non-private annotated method headers `ANH` contained in `ANHS` which are not accessible from `RT`;

9. $abs(\texttt{ANHS})$ returns all the abstract annotated method headers `ANH` contained in `ANHS`;

10. $nonAbs(\texttt{ANHS})$ returns all the non-abstract annotated method headers `ANH` contained in `ANHS`;

11. $ann(\texttt{RT},\texttt{NHS})$ returns the annotated method headers obtained by prefixing all the headers `NH` in `NHS` with the type `RT`.

As strange as it may seem, some of these functions distinguish between private and non-private members, even when using accessibility levels. In our model, which considers only top-level classes, private members are never accessible from classes different than the declaring one, so this extra-level of specification is redundant, except when selecting inaccessible members (which can be either private or non-private). Even if this is useful only in a single case, discussed below, we decided to model this extra-level consistently.

This is due to the fact that private members, even if accessible, are *never* inherited – JLS 8.2. For instance, inner classes *can* access private members of other inner classes (declared inside the same outer), but they do not inherit those members even when one inner class extends the other.

Figure 4.9 shows three metarules for deriving the *full types* of classes and interfaces. The full type of a reference type `RT` consists of three components:

- member fields: the set of annotated field signatures of all the fields that are inherited (and not hidden) by `RT` and of the fields directly declared by `RT`;
- member methods: the set of annotated method headers of all the methods that are inherited (and not overridden/hidden) by `RT` and of the methods directly declared by `RT`;
- non-overridden/hidden methods: the set of all the methods declared by superclasses and not yet overridden/hidden.

The first two components are the members of `RT`, as defined in JLS 6.4, and are used to resolve field accesses and method invocations; the third component, instead, is used only for technical reasons and is always empty for interfaces. Among non-overridden/hidden methods there are "ghost" methods; that is, methods that are not inherited (because they are not accessible), and that will never become members. These methods are to be taken into account because they may become accessible in subclasses, and affect the typechecking. The point is that inherited members of a class `C` are the non-private *accessible* members of the *direct* superclass $\mathtt{C}'$ – JLS 8.2 – but a method declared in `C` can override a method declared in a superclass $\mathtt{C}''$ (different from $\mathtt{C}'$) which is *not* a member of $\mathtt{C}'$ – JLS 8.4.6.1. Consider, for instance, the following example:

```
package p1 ;
public class A {
   int answer() { return 42 ; }
}

package p2 ;
public class B extends p1.A {}

package p1 ;
class C extends p2.B {
   double answer() { return 42.0 ; } // error: bad overriding of A.answer()
}
```

$$\frac{}{\Gamma^{\mathrm{S}} \vdash \texttt{Object} : \langle {\scriptstyle \mathrm{AFSS}=}\emptyset, {\scriptstyle \mathrm{ANHS}=}\emptyset, {\scriptstyle \mathrm{NOVR}=}\emptyset \rangle}$$

$$\frac{\begin{array}{l} \Gamma^{\mathrm{S}}(\texttt{I}).{\scriptstyle \mathrm{TYPE}} = \texttt{interface} \\ \Gamma^{\mathrm{S}}(\texttt{I}).{\scriptstyle \mathrm{AM}} = \texttt{AM} \\ \Gamma^{\mathrm{S}}(\texttt{I}).{\scriptstyle \mathrm{FSS}} = \texttt{FSS} \\ \Gamma^{\mathrm{S}}(\texttt{I}).{\scriptstyle \mathrm{NHS}} = \texttt{NHS} \\ \Gamma^{\mathrm{S}}(\texttt{I}).{\scriptstyle \mathrm{IS}} = \texttt{I}_1 \ldots \texttt{I}_n \\ \forall j \in \{1, \ldots, n\} \\ \quad \left\{ \begin{array}{l} \Gamma^{\mathrm{S}} \vdash \texttt{I} \not\lessdot \exists_{\texttt{ifc}}\, \texttt{I}_j \\ \Gamma^{\mathrm{S}} \vdash \texttt{I}_j : \langle {\scriptstyle \mathrm{AFSS}=}\texttt{AFSS}_j, {\scriptstyle \mathrm{ANHS}=}\texttt{ANHS}_j, {\scriptstyle \mathrm{NOVR}=}\emptyset \rangle \end{array} \right. \end{array}}{\begin{array}{ll} \Gamma^{\mathrm{S}} \vdash \texttt{I} : \langle & {\scriptstyle \mathrm{AFSS}=}(\bigcup_{i\in\{1,\ldots,n\}} \texttt{AFSS}_i)[\texttt{AFSS}]_{\Gamma^{\mathrm{S}}}, \\ & {\scriptstyle \mathrm{ANHS}=}(\bigoplus_{i\in\{1,\ldots,n\}} \texttt{ANHS}_i)[\texttt{ANHS}]_{\Gamma^{\mathrm{S}}}, \\ & {\scriptstyle \mathrm{NOVR}=}\emptyset \\ & \rangle \end{array}} \quad \begin{array}{l} \texttt{AFSS} = \mathit{ann}(\texttt{I}, \texttt{FSS}) \\ \texttt{ANHS} = \mathit{ann}(\texttt{I}, \texttt{NHS}) \\ \texttt{AM} \in \{\epsilon, \texttt{public}\} \\ \forall \texttt{FS} \in \texttt{FSS} : \\ \quad \left\{ \begin{array}{l} \mathit{accessModifier}(\texttt{FS}) = \texttt{public} \\ \mathit{fieldKind}(\texttt{FS}) = \texttt{static} \\ \mathit{finalModifier}(\texttt{FS}) = \texttt{final} \end{array} \right. \\ \forall \texttt{NH} \in \texttt{NHS} : \\ \quad \left\{ \begin{array}{l} \mathit{accessModifier}(\texttt{NH}) = \texttt{public} \\ \mathit{methodKind}(\texttt{FS}) = \texttt{abstract} \end{array} \right. \end{array}$$

$$\frac{\begin{array}{l} \Gamma^{\mathrm{S}}(\texttt{C}).{\scriptstyle \mathrm{AM}} = \texttt{AM} \\ \Gamma^{\mathrm{S}}(\texttt{C}).{\scriptstyle \mathrm{CK}} = \texttt{CK} \\ \Gamma^{\mathrm{S}}(\texttt{C}).{\scriptstyle \mathrm{PARENT}} = \texttt{C}' \\ \Gamma^{\mathrm{S}} \vdash \texttt{C} \not\lessdot \exists_{\texttt{cls}}\, \texttt{any}\ \texttt{C}' \\ \Gamma^{\mathrm{S}} \vdash \texttt{C}' : \langle {\scriptstyle \mathrm{AFSS}=}\texttt{AFSS}_0, {\scriptstyle \mathrm{ANHS}=}\texttt{ANHS}_0, {\scriptstyle \mathrm{NOVR}=}\texttt{ANHS}_{\texttt{NO}} \rangle \\ \widetilde{\texttt{AFSS}_0} = \mathit{accNP}_{\Gamma^{\mathrm{S}}}(\texttt{C}, \texttt{AFSS}_0) \\ \widetilde{\texttt{ANHS}_0} = \mathit{accNP}_{\Gamma^{\mathrm{S}}}(\texttt{C}, \texttt{ANHS}_0) \\ \Gamma^{\mathrm{S}}(\texttt{C}).{\scriptstyle \mathrm{KSS}} = \texttt{KSS} \\ \Gamma^{\mathrm{S}}(\texttt{C}).{\scriptstyle \mathrm{FSS}} = \texttt{FSS} \\ \Gamma^{\mathrm{S}}(\texttt{C}).{\scriptstyle \mathrm{NHS}} = \texttt{NHS} \\ \Gamma^{\mathrm{S}}(\texttt{C}).{\scriptstyle \mathrm{IS}} = \texttt{I}_1 \ldots \texttt{I}_n \\ \forall j \in \{1, \ldots, n\} \\ \quad \left\{ \begin{array}{l} \Gamma^{\mathrm{S}} \vdash \texttt{C} \not\lessdot \exists_{\texttt{ifc}}\, \texttt{I}_j \\ \Gamma^{\mathrm{S}} \vdash \texttt{I}_j : \langle {\scriptstyle \mathrm{AFSS}=}\texttt{AFSS}_j, {\scriptstyle \mathrm{ANHS}=}\texttt{ANHS}_j, {\scriptstyle \mathrm{NOVR}=}\emptyset \rangle \end{array} \right. \end{array}}{\begin{array}{ll} \Gamma^{\mathrm{S}} \vdash & \texttt{C} : \langle {\scriptstyle \mathrm{AFSS}=}(\widetilde{\texttt{AFSS}_0} \cup (\bigcup_{i\in\{1,\ldots,n\}} \texttt{AFSS}_i))[\texttt{AFSS}]_{\Gamma^{\mathrm{S}}}, \\ & {\scriptstyle \mathrm{ANHS}=}(\mathit{abs}(\widetilde{\texttt{ANHS}_0}) \oplus (\bigoplus_{i\in\{1,\ldots,n\}} \texttt{ANHS}_i))[\mathit{nonAbs}(\widetilde{\texttt{ANHS}_0})[\texttt{ANHS}]_{\Gamma^{\mathrm{S}}}]_{\Gamma^{\mathrm{S}}}, \\ & {\scriptstyle \mathrm{NOVR}=}\ \mathit{accNP}_{\Gamma^{\mathrm{S}}}(\texttt{C}, \texttt{ANHS}_{\texttt{NO}})[\mathit{nonPrivate}(\texttt{ANHS})]_{\Gamma^{\mathrm{S}}} \cup \\ & \qquad \mathit{nonAccNP}_{\Gamma^{\mathrm{S}}}(\texttt{C}, \texttt{ANHS}_{\texttt{NO}}) \\ & \rangle \end{array}} \quad \begin{array}{l} \texttt{KSS} \neq \emptyset \\ \texttt{AFSS} = \mathit{ann}(\texttt{C}, \texttt{FSS}) \\ \texttt{ANHS} = \mathit{ann}(\texttt{C}, \texttt{NHS}) \\ \texttt{AM} \in \{\epsilon, \texttt{public}\} \\ \texttt{CK} = \epsilon \implies \\ \mathit{abs}(\texttt{ANHS} \cup \texttt{ANHS}_{\texttt{NO}}) \\ \qquad = \emptyset \end{array}$$

Figure 4.9: Full types

$$\mathtt{ANHS}_1 \oplus \mathtt{ANHS}_2 = \begin{cases} \mathtt{ANHS}_1 \cup \mathtt{ANHS}_2 & \text{if } \forall \mathtt{ANH}_1 \in \mathtt{ANHS}_1, \mathtt{ANH}_2 \in \mathtt{ANHS}_2 : \\ & \quad \mathit{nAndP}(\mathtt{ANH}_1) = \mathit{nAndP}(\mathtt{ANH}_2) \implies \\ & \qquad \mathit{ret}(\mathtt{ANH}_1) = \mathit{ret}(\mathtt{ANH}_2) \\ \bot & \text{otherwise} \end{cases}$$

$$\mathtt{ANHS}[\{\mathtt{ANH}_1, \ldots, \mathtt{ANH}_n\}]_{\Gamma^{\mathrm{S}}} = \mathtt{ANHS}[\mathtt{ANH}_1]_{\Gamma^{\mathrm{S}}} \ldots [\mathtt{ANH}_n]_{\Gamma^{\mathrm{S}}}$$

$$\mathtt{ANHS}[\mathtt{RT}'.\mathtt{NH}']_{\Gamma^{\mathrm{S}}} = \begin{cases} \{\mathtt{RT.NH}'\} \cup \{\mathtt{ANH} : \mathtt{ANH} \in \mathtt{ANHS}, & \text{if } \forall \mathtt{RT.NH} \in \mathtt{ANHS} : \\ \quad \mathit{nAndP}(\mathtt{ANH}) \neq \mathit{nAndP}(\mathtt{ANH}')\} & \quad \mathit{nAndP}(\mathtt{NH}) = \mathit{nAndP}(\mathtt{NH}') \implies \\ & \qquad \Gamma^{\mathrm{S}} \vdash \mathtt{NH}' \leq \mathtt{NH} \\ \bot & \text{otherwise} \end{cases}$$

$$\mathtt{AFSS}[\{\mathtt{AFS}_1, \ldots, \mathtt{AFS}_n\}]_{\Gamma^{\mathrm{S}}} = \mathtt{ANHS}[\mathtt{AFS}_1]_{\Gamma^{\mathrm{S}}} \ldots [\mathtt{AFS}_n]_{\Gamma^{\mathrm{S}}}$$
$$\mathtt{AFSS}[\mathtt{AFS}']_{\Gamma^{\mathrm{S}}} = \{\mathtt{AFS}'\} \cup \{\mathtt{AFS} \in \mathtt{AFSS} : \mathit{name}(\mathtt{AFS}) \neq \mathit{name}(\mathtt{AFS}')\}$$

$$\mathit{nAndP}(\mathtt{RT.NH}) = \mathit{nAndP}(\mathtt{NH})$$
$$\mathit{nAndP}(\mathtt{AM\ MK\ T\ m}(\bar{\mathtt{T}})\ \mathtt{throws\ ES}) = \mathtt{m}(\bar{\mathtt{T}})$$
$$\mathit{ret}(\mathtt{RT.AM\ MK\ T\ m}(\bar{\mathtt{T}})\ \mathtt{throws\ ES}) = \mathtt{T}$$
$$\mathit{name}(\mathtt{RT.AM\ FINAL\ FK\ T\ f}) = \mathtt{f}$$
$$\mathit{name}(\mathtt{RT.AM\ final\ static\ T\ f}\ =\ \nu) = \mathtt{f}$$
$$\mathit{accessModifier}(\mathtt{AM\ FINAL\ FK\ T\ f}) = \mathtt{AM}$$
$$\mathit{accessModifier}(\mathtt{AM\ final\ static\ T\ f}\ =\ \nu) = \mathtt{AM}$$
$$\mathit{accessModifier}(\mathtt{AM\ MK\ T\ m}(\bar{\mathtt{T}})\ \mathtt{throws\ ES}) = \mathtt{AM}$$

$$\mathit{fieldKind}(\mathtt{AM\ FINAL\ FK\ T\ f}) = \mathtt{FK}$$
$$\mathit{fieldKind}(\mathtt{AM\ final\ static\ T\ f}\ =\ \nu) = \mathtt{FK}$$

$$\mathit{finalModifier}(\mathtt{AM\ FINAL\ FK\ T\ f}) = \mathtt{FINAL}$$
$$\mathit{finalModifier}(\mathtt{AM\ final\ static\ T\ f}\ =\ \nu) = \mathtt{final}$$

$$\mathit{methodKind}(\mathtt{AM\ MK\ T\ m}(\bar{\mathtt{T}})\ \mathtt{throws\ ES}) = \mathtt{MK}$$

Figure 4.10: Auxiliary operators and functions

In this example class p1.A declares a method named answer which returns an int. This method, having a default access, is not accessible in package p2 so the method is not a member of class p2.B (which has no members). Class p1.C, extending p2.B, inherits all its members (that is, none) and declares a method named answer which returns a double. However, being A and C declared in the same package, the method A.answer is accessible from C (albeit code inside p1 *cannot* invoke it through objects of type C because answer is not inherited) and, so, the requirements on method overriding must be met by int answer() and double answer(). In this example, of course, they are not, because the return type of the latter method differs from the return type of the former.

In order to check the requirements on overriding and hiding – JLS 8.4.6.3 – and to check whether a class has abstract methods – JLS 8.1.1.1 – we need to keep track of all the methods (declared in superclasses) which are not overridden. Indeed, the presence of a ghost abstract method m prevents a class to be declared non-abstract, albeit m is "invisible".

Because all members are public in interfaces, all members are inherited by subinterfaces therefore we do not need to keep track of ghost methods as we do with classes.

The metarules in Figure 4.9 use some auxiliary operators which are defined in Figure 4.10:

- Operator $\oplus$ is used to merge sets of annotated method headers; it corresponds to set union, apart that it is only defined when there are no two different method headers with the same name and parameter types but different return type. This restriction is due to the fact that a class (or an interface) can inherit more than one method with the same name and parameter types (from the superclass and/or superinterfaces) as long as all the methods have the same return type – JLS 8.4.6.4 for classes and JLS 9.4.1 for interfaces.

- Operator $\_[\_]_{\Gamma^S}$ is used to encode overriding/hiding on both methods and fields. That is, $\mathtt{ANHS}[\mathtt{ANHS}']_{\Gamma^S}$ corresponds to the set union of the annotated method headers in $\mathtt{ANHS}'$ with the annotated method headers in ANHS which are not overridden/hidden by a header in $\mathtt{ANHS}'$. This operation is defined only when the requirements on method overriding/hiding are met. Analogously, $\mathtt{AFSS}[\mathtt{AFSS}']_{\Gamma^S}$ corresponds to the set union of the annotated field signatures in $\mathtt{AFSS}'$ with the annotated field signatures in AFSS which are not hidden by a signature in $\mathtt{AFSS}'$. This operation is always defined because there are no requirements on hiding of fields.

The full type of Object (first metarule in Figure 4.9) simply consists of three empty sets as, for simplicity, we ignore all the predefined methods of Object (defined in JLS 4.3.2).

The members of an interface I consist of all the declared fields and methods, annotated with I, plus all the (annotated) members of its superinterfaces which have not been hidden by those declared in I. So, the full type of an interface consists of the member fields, the member methods and, as said before, an empty set.

$$
\frac{\Gamma^{\mathrm{S}}(\mathtt{C}).{\scriptstyle \mathrm{KSS}} = \mathtt{KSS} \qquad \mathit{mostSpec}_{\Gamma^{\mathrm{S}}}(\mathit{acc}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{KSS}), \bar{\mathtt{T}}^{\perp}) = \{\mathtt{AM}\ \bar{\mathtt{T}}\ \mathtt{throws}\ \mathtt{ES}\}}{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \sphericalangle \mathit{Cns}(\mathtt{C}, \bar{\mathtt{T}}^{\perp}) = [{\scriptstyle \mathrm{PAR}=}\bar{\mathtt{T}}]}
$$

$$
\begin{aligned}
&\mathit{mostSpec}_{\Gamma^{\mathrm{S}}}(\mathtt{KSS}, \bar{\mathtt{T}}^{\perp}) = \{\mathtt{KS} \mid \mathtt{KS} \in \mathit{appl}_{\Gamma^{\mathrm{S}}}(\mathtt{KSS}, \bar{\mathtt{T}}^{\perp}), \\
&\qquad\qquad \forall \mathtt{KS}' \in \mathit{appl}_{\Gamma^{\mathrm{S}}}(\mathtt{KSS}, \bar{\mathtt{T}}^{\perp}), \Gamma^{\mathrm{S}} \vdash \mathit{params}(\mathtt{KS}) \leq \mathit{params}(\mathtt{KS}')\} \\
&\mathit{appl}_{\Gamma^{\mathrm{S}}}(\mathtt{KSS}, \bar{\mathtt{T}}^{\perp}) = \{\mathtt{KS} \mid \mathtt{KS} \in \mathtt{KSS}, \Gamma^{\mathrm{S}} \vdash \bar{\mathtt{T}}^{\perp} \leq \mathit{params}(\mathtt{KS})\} \\
&\mathit{params}(\mathtt{AM}\ \bar{\mathtt{T}}\ \mathtt{throws}\ \mathtt{ES}) = \bar{\mathtt{T}}
\end{aligned}
$$

Figure 4.11: Constructors

The full type of a class `C` is the most complex to calculate, so let us describe the components one by one. The first component, corresponding to fields, is the easiest to calculate: the member fields consists of all the declared fields, annotated with `C`, plus the accessible fields inherited from the superclass and all the fields inherited from the superinterfaces (they are always accessible) which have not been hidden by a field declared in the class.

The second component, corresponding to methods, is rather complex so let us consider abstract and non-abstract methods separately. The inherited abstract methods are the (non-private) accessible ones inherited from the superclass $\mathtt{C}'$ and all the members of the superinterfaces $\mathtt{I}_j$ which have not been overridden by a declaration in the class `C` *or* by a non-abstract member inherited from the superclass – JLS 8.4.6.4 (note that in this case the method is not declared in `C`, otherwise the one declared in `C` would override them all). To the set of these abstract methods we must add the (non-private) accessible non-abstract methods inherited from $\mathtt{C}'$, which have not been overridden by a declaration in `C`, and, finally, all the methods declared in the class (annotated with `C`).

The third component must keep track, as said, of all declared methods not yet overridden. This component consists of:

- all the non-private methods declared in the class[9];
- all the accessible not overridden methods which are still not overridden;
- all the non-accessible methods not overridden (which, being inaccessible, cannot be overridden by any method in `C`).

Figure 4.11 show the metarule for deriving which is the most specific constructor for a constructor invocation (via `super` or `new`). Because constructors are never inherited, all

[9]Technically, every method is overridden by itself but here we consider only proper overriding; that is, overriding by a method defined in a proper subclass.

the information about the constructors of a class `C` can be found looking at `C` only. An invocation is correct only if there is a unique most specific constructor that is both applicable and accessible – JLS 15.9.3.

Figure 4.12 show the metarules for resolving field accesses and for evaluating constant expressions. This latter metarule uses the auxiliary function *eval*, defined in the same figure. A field access is successful resolved only when there is only one field which is accessible. If there are no accessible fields then the field access is undefined and, conversely, if there are more than one field then the access is ambiguous – JLS 15.11.1. In our model the judgment for resolving a field access can be derived only when the field is *not* a ctc-field. That is, when the field is not static or final (first metarule) or when, despite being both static and final, its initialization expression cannot be evaluated at compile-time (second metarule). This choice reflects the fact that a field access expression is compiled to a field access binary expression only if the field is not a ctc-field. When it is, as the reader may recall, the field access expression must be directly compiled to the (constant) value of the field.

Figure 4.13 show the metarules for resolving method invocations. An invocation is successfully resolved if a single most specific method can be found among the maximally specific ones. A method is said to be maximally specific for a method invocation if it is applicable, accessible and there is no other applicable and accessible method that is more specific; see the definition of *maxSpec*. If there is only one maximally specific method, then it is the most specific one – JLS 15.12.2.2; in the definition of *mostSpec*, this case is seen as a special case of the other following subcase:

- if all the maximally specific methods have the same name and parameter types:
  - if one is not abstract (this includes the case where there is just one non-abstract method in `ANHS`), then it is the most specific (there cannot be more than one non-abstract methods since non-abstract methods can only be inherited from superclasses);
  - if they are all abstract (this includes the case where there is just one abstract method in `ANHS`), then the most specific method is chosen arbitrarily among the maximally specific methods. However, the most specific method is considered to throw an exception if and only if that exception has a superclass in the throws clause of each of the maximally specific methods.
- Otherwise, the method invocation is ambiguous.

Figure 4.14 show the metarules for evaluating exception expressions ((1) and (2)); they use an auxiliary judgment $\Gamma^{\mathrm{S}} \vdash \texttt{EE} \rightarrow_{\texttt{Exc}} \texttt{ES}$ which can be derived when the exception expression `EE` is equivalent to the set of exception (that is, class) names `ES` in Γ^{S}.

$$\frac{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \mathtt{RT}' : \langle\text{AFSS=}\mathtt{AFSS}, \text{ANHS=}\_, \text{NOVR=}\_\rangle \\ \mathit{mostSpec}_{\Gamma^{\mathrm{S}}}(\mathit{acc}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{AFSS}), \mathtt{f}) = \{\mathtt{RT}''.\mathtt{AM\ FINAL\ FK\ T\ f}\ \}\end{array}}{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\leqslant \mathit{Fld}(\mathtt{RT}', \mathtt{f}) = [\text{FINAL=}\mathtt{FINAL}, \text{FK=}\mathtt{FK}, \text{T=}\mathtt{T}] \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\leqslant \mathit{Fld}(\mathtt{RT}', \mathtt{f}) = [\text{FINAL=}\mathtt{any}, \text{FK=}\mathtt{FK}, \text{T=}\mathtt{T}]\end{array}}$$

$$\frac{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \mathtt{RT}' : \langle\text{AFSS=}\mathtt{AFSS}, \text{ANHS=}\_, \text{NOVR=}\_\rangle \\ \mathit{mostSpec}_{\Gamma^{\mathrm{S}}}(\mathit{acc}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{AFSS}), \mathtt{f}) = \{\mathtt{RT}''.\mathtt{AM\ final\ static\ T\ f}\ = \mathtt{E}^s\ \} \\ \mathit{eval}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{E}^s, \emptyset) = \bot\end{array}}{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\leqslant \mathit{Fld}(\mathtt{RT}', \mathtt{f}) = [\text{FINAL=}\mathtt{FINAL}, \text{FK=}\mathtt{FK}, \text{T=}\mathtt{T}] \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\leqslant \mathit{Fld}(\mathtt{RT}', \mathtt{f}) = [\text{FINAL=}\mathtt{any}, \text{FK=}\mathtt{FK}, \text{T=}\mathtt{T}]\end{array}}$$

$$\begin{array}{l}\mathit{mostSpec}_{\Gamma^{\mathrm{S}}}(\mathtt{AFSS}, \mathtt{f}) = \{\mathtt{AFS} |\ \mathtt{AFS} \in \mathit{appl}_{\Gamma^{\mathrm{S}}}(\mathtt{AFSS}, \mathtt{f}), \\ \qquad\qquad\qquad\qquad\qquad \forall \mathtt{AFS}' \in \mathit{appl}_{\Gamma^{\mathrm{S}}}(\mathtt{AFSS}, \mathtt{f}) : \Gamma^{\mathrm{S}} \vdash \mathtt{AFS} \leq \mathtt{AFS}'\} \\ \mathit{appl}_{\Gamma^{\mathrm{S}}}(\mathtt{AFSS}, \mathtt{f}) = \{\mathtt{AFS} | \mathtt{AFS} \in \mathtt{AFSS}, \mathit{name}(\mathtt{AFS}) = \mathtt{f}\} \\ \mathit{name}(\mathtt{AM\ FINAL\ FK\ T\ f} \ldots) = \mathtt{f}\end{array}$$

$$\mathit{eval}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{E}^s, \mathit{pending}) = \begin{cases} \nu & \text{if } \ \mathtt{E}^s = \nu \\ \iota_1 \,\Theta\, \iota_2 & \text{if } \ \mathtt{E}^s = \mathtt{E}^s_1 \,\Theta\, \mathtt{E}^s_2 \\ & \quad \Theta \in \{+, -, \ldots\} \\ & \quad \mathit{eval}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{E}^s_1, \mathit{pending}) = \iota_1 \\ & \quad \mathit{eval}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{E}^s_2, \mathit{pending}) = \iota_2 \\ \nu & \text{if } \ \mathtt{E}^s = \mathtt{RT}'.\mathtt{f} \\ & \quad \Gamma^{\mathrm{S}} \vdash \mathtt{RT}' : \langle\text{AFSS=}\_, \text{ANHS=}\mathtt{AFSS}, \text{NOVR=}\_\rangle \\ & \quad \mathit{mostSpec}_{\Gamma^{\mathrm{S}}}(\mathit{acc}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{AFSS}), \mathtt{f}) = \\ & \qquad \{\mathtt{RT}''.\mathtt{AM\ final\ static\ T\ f}\ =\ \mathtt{E}^s_{\mathtt{init}}\ \} \\ & \quad \mathtt{RT}''.\mathtt{f} \notin \mathit{pending} \\ & \quad \mathit{eval}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}'', \mathtt{E}^s_{\mathtt{init}}, \mathit{pending} \cup \{\mathtt{RT}''.\mathtt{f}\}) = \nu \\ \bot & \text{otherwise} \end{cases}$$

$$\frac{\mathit{eval}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{E}^s, \emptyset) = \nu}{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\leqslant \mathtt{E}^s = \nu}$$

Figure 4.12: Fields and constants

$$\frac{\begin{array}{l}\Gamma^{\mathtt{S}} \vdash \mathtt{RT}' : \langle \textsc{afss}{=}\_,\ \textsc{anhs}{=}\mathtt{ANHS},\ \textsc{novr}{=}\_\rangle \\ mostSpec_{\Gamma^{\mathtt{S}}}(acc_{\Gamma^{\mathtt{S}}}(\mathtt{RT},\mathtt{ANHS}),\mathtt{m}(\bar{\mathtt{T}}^{\perp})) = \mathtt{AM\ static\ T\ m}(\bar{\mathtt{T}})\ \mathtt{throws\ ES}\end{array}}{\mathtt{RT} \measuredangle Mth(\mathtt{RT}',\mathtt{m},\bar{\mathtt{T}}^{\perp}) = [\textsc{mk}{=}\mathtt{static},\ \textsc{ret}{=}\mathtt{T},\ \textsc{par}{=}\bar{\mathtt{T}}]}$$

$$\frac{\begin{array}{l}\Gamma^{\mathtt{S}} \vdash \mathtt{RT}' : \langle \textsc{afss}{=}\_,\ \textsc{anhs}{=}\mathtt{ANHS},\ \textsc{novr}{=}\_\rangle \\ mostSpec_{\Gamma^{\mathtt{S}}}(acc_{\Gamma^{\mathtt{S}}}(\mathtt{RT},\mathtt{ANHS}),\mathtt{m}(\bar{\mathtt{T}}^{\perp})) = \mathtt{AM\ MK\ T\ m}(\bar{\mathtt{T}})\ \mathtt{throws\ ES}\end{array}}{\mathtt{RT} \measuredangle Mth(\mathtt{RT}',\mathtt{m},\bar{\mathtt{T}}^{\perp}) = [\textsc{mk}{=}\mathtt{not\text{-}static},\ \textsc{ret}{=}\mathtt{T},\ \textsc{par}{=}\bar{\mathtt{T}}]}\ \mathtt{MK} \in \{\epsilon, \mathtt{abstract}\}$$

$$mostSpec_{\Gamma^{\mathtt{S}}}(\mathtt{ANHS},\mathtt{m}(\bar{\mathtt{T}}^{\perp})) =$$
$$\begin{cases} \mathtt{ANH} & \text{if } maxSpec_{\Gamma^{\mathtt{S}}}(\mathtt{ANHS},\mathtt{m}(\bar{\mathtt{T}}^{\perp})) = \mathtt{ANHS}' \\ & \quad \forall \mathtt{ANH}_1,\mathtt{ANH}_2 \in \mathtt{ANHS}' : nAndP(\mathtt{ANH}_1) = nAndP(\mathtt{ANH}_2) \\ & \quad nonAbs(\mathtt{ANHS}') = \{\mathtt{ANH}\} \\ \perp.\mathtt{public\ abstract} & \text{if } maxSpec_{\Gamma^{\mathtt{S}}}(\mathtt{ANHS},\mathtt{m}(\bar{\mathtt{T}}^{\perp})) = \mathtt{ANHS}' \\ \mathtt{T\ m}(\bar{\mathtt{T}})\ \mathtt{throws\ ES} & \quad \mathtt{ANHS}' \neq \emptyset \\ & \quad \forall \mathtt{ANH}_1,\mathtt{ANH}_2 \in \mathtt{ANHS}' : nAndP(\mathtt{ANH}_1) = nAndP(\mathtt{ANH}_2) \\ & \quad nonAbs(\mathtt{ANHS}') = \emptyset \\ & \quad \mathtt{T} = ret(\mathtt{ANH}'),\ \text{for any } \mathtt{ANH}' \in \mathtt{ANHS}' \\ & \quad \mathtt{ES} = \{\mathtt{C} \mid \forall \mathtt{ANH}' \in \mathtt{ANHS}'\ \exists \mathtt{C}' \in throws(\mathtt{ANH}') : \Gamma^{\mathtt{S}} \vdash \mathtt{C} \leq \mathtt{C}'\} \\ \perp & \text{otherwise} \end{cases}$$

$$\begin{array}{rl} maxSpec_{\Gamma^{\mathtt{S}}}(\mathtt{ANHS},\mathtt{m}(\bar{\mathtt{T}}^{\perp})) = \{\mathtt{ANH} \mid & \mathtt{ANH} \in appl_{\Gamma^{\mathtt{S}}}(\mathtt{ANHS},\mathtt{m}(\bar{\mathtt{T}}^{\perp})), \\ & \forall \mathtt{ANH}' \in appl_{\Gamma^{\mathtt{S}}}(\mathtt{ANHS},\mathtt{m}(\bar{\mathtt{T}}^{\perp})) : \\ & \Gamma^{\mathtt{S}} \vdash \mathtt{ANH}' \leq \mathtt{ANH} \implies \mathtt{ANH} = \mathtt{ANH}' \} \end{array}$$

$$appl_{\Gamma^{\mathtt{S}}}(\mathtt{ANHS},\mathtt{m}(\bar{\mathtt{T}}^{\perp})) = \{\mathtt{ANH} \mid \mathtt{ANH} \in \mathtt{ANHS},\ nAndP(\mathtt{ANH}) = \mathtt{m}(\bar{\mathtt{T}}), \Gamma^{\mathtt{S}} \vdash \bar{\mathtt{T}}^{\perp} \leq \bar{\mathtt{T}}\}$$

$$throws(\mathtt{RT.AM\ MK\ T\ m}(\bar{\mathtt{T}})\ \mathtt{throws\ ES}) = \mathtt{ES}$$

Figure 4.13: Methods

$$(1)\ \frac{\Gamma^{\mathrm{S}} \vdash \mathtt{EE} \rightarrow_{\mathtt{Exc}} \mathtt{ES}'}{\Gamma^{\mathrm{S}} \vdash \mathtt{EE} \supseteq_{\mathtt{Exc}} \mathtt{ES}} \qquad \forall \mathtt{C} \in \mathtt{ES}\ \exists \mathtt{C}' \in \mathtt{ES}' \,:\, \Gamma^{\mathrm{S}} \vdash \mathtt{C} \leq \mathtt{C}'$$

$$(2)\ \frac{\Gamma^{\mathrm{S}} \vdash \mathtt{EE} \rightarrow_{\mathtt{Exc}} \mathtt{ES}'}{\Gamma^{\mathrm{S}} \vdash \mathtt{EE} \subseteq_{\mathtt{Exc}} \mathtt{ES}} \qquad \forall \mathtt{C}' \in \mathtt{ES}'\ \exists \mathtt{C} \in \mathtt{ES} \,:\, \Gamma^{\mathrm{S}} \vdash \mathtt{C}' \leq \mathtt{C}$$

$$(3)\ \frac{\begin{array}{l}\Gamma^{\mathrm{S}}(\mathtt{C}).\textsc{kss} = \mathtt{KSS} \\ \mathtt{AM}\ \bar{\mathtt{T}}\ \mathtt{throws}\ \mathtt{ES} \in acc_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{KSS})\end{array}}{\Gamma^{\mathrm{S}} \vdash \varepsilon(\mathtt{RT} \not\lessdot \mathtt{C}(\bar{\mathtt{T}})) \rightarrow_{\mathtt{Exc}} \mathtt{ES}}$$

$$(4)\ \frac{\begin{array}{l}\Gamma^{\mathrm{S}} \vdash \mathtt{RT}' : \langle \textsc{afss}{=}\_, \textsc{anhs}{=}\mathtt{ANHS}, \textsc{novr}{=}\_\rangle \\ mostSpec_{\Gamma^{\mathrm{S}}}(acc_{\Gamma^{\mathrm{S}}}(\mathtt{RT}, \mathtt{ANHS}), \mathtt{m}(\bar{\mathtt{T}}^{\perp})) = \mathtt{AM}\ \mathtt{MK}\ \mathtt{T}\ \mathtt{m}(\bar{\mathtt{T}})\ \mathtt{throws}\ \mathtt{ES}\end{array}}{\Gamma^{\mathrm{S}} \vdash \varepsilon(\mathtt{RT} \not\lessdot \mathtt{RT}'.\mathtt{m}(\bar{\mathtt{T}}^{\perp})) \rightarrow_{\mathtt{Exc}} \mathtt{ES}}$$

$$(5)\ \frac{}{\Gamma^{\mathrm{S}} \vdash \mathtt{ES} \rightarrow_{\mathtt{Exc}} \mathtt{ES}}$$

$$(6)\ \frac{\Gamma^{\mathrm{S}} \vdash \mathtt{EE}_1 \rightarrow_{\mathtt{Exc}} \mathtt{ES}_1 \quad \Gamma^{\mathrm{S}} \vdash \mathtt{EE}_2 \rightarrow_{\mathtt{Exc}} \mathtt{ES}_2}{\Gamma^{\mathrm{S}} \vdash \mathtt{EE}_1 \cup_{\mathtt{Exc}} \mathtt{EE}_2 \rightarrow_{\mathtt{Exc}} \mathtt{ES}_1 \cup \mathtt{ES}_2}$$

$$(7)\ \frac{\Gamma^{\mathrm{S}} \vdash \mathtt{EE}_1 \rightarrow_{\mathtt{Exc}} \mathtt{ES}_1 \quad \Gamma^{\mathrm{S}} \vdash \mathtt{EE}_2 \rightarrow_{\mathtt{Exc}} \mathtt{ES}_2}{\Gamma^{\mathrm{S}} \vdash \mathtt{EE}_1 \setminus_{\mathtt{Exc}} \mathtt{EE}_2 \rightarrow_{\mathtt{Exc}} \{\mathtt{C}_1 \in \mathtt{ES}_1 \mid \nexists \mathtt{C}_2 \in \mathtt{ES}_2 : \Gamma^{\mathrm{S}} \vdash \mathtt{C}_1 \leq \mathtt{C}_2\}}$$

Figure 4.14: Rules defining exception related judgments

Metarule (3) evaluates the exceptions that may be thrown by an invocation, inside $\mathtt{RT}$, of the constructor of class $\mathtt{C}$ with parameter types $\bar{\mathtt{T}}$.

Metarule (4) evaluates, using the auxiliary function *mostSpec* defined in Figure 4.13, the exceptions that may be thrown by an invocation, inside $\mathtt{RT}$, of a method named $\mathtt{m}$ with parameter types $\bar{\mathtt{T}}^{\perp}$. Metarules (5) and (6) are trivial; they respectively state that a set of exception names $\mathtt{ES}$ evaluates to itself, and that the union of two exception expressions evaluates to the union of their respective sets of exception names. The last metarule, (7), is more interesting: the "difference" between two sets of exceptions is defined as the set of exception names $\mathtt{C}_1$ which are contained in the first set $\mathtt{ES}_1$, such that they cannot be captured by any exception $\mathtt{C}_2$ contained in the second set $\mathtt{ES}_2$. That is, such that there is none of their superclasses in $\mathtt{ES}_2$. An in-depth discussion of these issues, regarding the formalization of Java exceptions, can be found in [ALZ01].

The next subsection shows how and when these assumptions are used in the process of compilation.

4.2.4 Compilation

Compilation of expressions is expressed by the following judgment:

$$\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}^s \leadsto \mathtt{E}^b : \mathtt{T}\ \textbf{throws:}\ \mathtt{EE}$$

with the meaning "expression $\mathtt{E}^s$ has type $\mathtt{T}$, throws exceptions $\mathtt{EE}$ and compiles to binary expression $\mathtt{E}^b$ when contained in type $\mathtt{RT}$, in a type environment Γ^{S} and local environment Π". Type $\mathtt{RT}$ is needed to model the access control; for instance, `private` methods of $\mathtt{RT}$ can be invoked only by expressions inside $\mathtt{RT}$. The local environment Π maps parameter names and `this` to their respective types (`this` is undefined when typing expressions contained in static contexts).

Figures 4.15 to 4.17 show the metarules defining this judgment.

Rules (1) and (2) in Figure 4.15 model the compilation of a compile-time constant expression of type, respectively, `int` and `bool`. As explained in Section 4.1.1, these expressions are compiled directly to their corresponding value and, obviously, their evaluation throws no exceptions (so $\mathtt{EE}$ is the empty set).

In rule (3) two expressions, which are not both compile-time constants, are combined using an operator Θ: the result is a non-constant expression that can throw any exception its operands can throw.

The literal `null` is compiled into itself, has type $\perp$, and throws no exception, metarule (4).

The keyword `this` is compiled into itself, has type $\Pi(\mathtt{this})$, and throws no exceptions,

$$(1)\quad \frac{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\prec \mathtt{E}^s = \iota}{\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}^s \leadsto \iota : \mathtt{int}\ \textbf{throws: } \emptyset}$$

$$(2)\quad \frac{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\prec \mathtt{E}^s = \beta}{\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}^s \leadsto \beta : \mathtt{bool}\ \textbf{throws: } \emptyset}$$

$$(3)\quad \frac{\begin{array}{c}\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_1^s \leadsto \mathtt{E}_1^b : \mathtt{int}\ \textbf{throws: } \mathtt{EE}_1 \\ \Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_2^s \leadsto \mathtt{E}_2^b : \mathtt{int}\ \textbf{throws: } \mathtt{EE}_2\end{array}}{\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_1^s \,\Theta\, \mathtt{E}_2^s \leadsto \mathtt{E}_1^b \,\Theta\, \mathtt{E}_2^b : \mathtt{int}\ \textbf{throws: } \mathtt{EE}_1 \cup_{\mathtt{Exc}} \mathtt{EE}_2} \qquad \begin{array}{l}\Theta \in \{+, -, \ldots\} \\ \mathtt{E}_1^b \text{ or } \mathtt{E}_2^b \text{ is not a value}\end{array}$$

$$(4)\quad \frac{}{\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{null} \leadsto \mathtt{null} : \bot\ \textbf{throws: } \emptyset}$$

$$(5)\quad \frac{}{\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{this} \leadsto \mathtt{this} : \Pi(\mathtt{this})\ \textbf{throws: } \emptyset}$$

$$(6)\quad \frac{}{\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{x} \leadsto \mathtt{x} : \Pi(\mathtt{x})\ \textbf{throws: } \emptyset}$$

$$(7)\quad \frac{\begin{array}{l}\forall i \in \{1, \ldots, n\}\ \Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_i^s \leadsto \mathtt{E}_i^b : \mathtt{T}_i\ \textbf{throws: } \mathtt{EE}_i \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\prec \exists_{\mathtt{cls}}\, \epsilon\ \mathtt{C} \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\prec \mathit{Cns}(\mathtt{C}, \mathtt{T}_1 \ldots \mathtt{T}_n) = [\textsc{par}{=}\bar{\mathtt{T}}]\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash & \mathtt{new}\ \mathtt{C}(\mathtt{E}_1^s, \ldots, \mathtt{E}_n^s) \leadsto \\ & \mathtt{new}\ \ll \mathtt{C}(\bar{\mathtt{T}}) \gg_{\mathtt{c}} (\mathtt{E}_1^b, \ldots, \mathtt{E}_n^b) : \mathtt{C} \\ & \quad \textbf{throws: } \mathtt{EE}_1 \cup_{\mathtt{Exc}} \ldots \cup_{\mathtt{Exc}} \mathtt{EE}_n \cup_{\mathtt{Exc}} \varepsilon(\mathtt{RT} \not\prec \mathtt{C}(\bar{\mathtt{T}}))\end{array}}$$

$$(8)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_0^s \leadsto \mathtt{E}_0^b : \mathtt{C}\ \textbf{throws: } \mathtt{EE}_0 \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\prec \exists_{\mathtt{cls}}\, \mathtt{any}\ \mathtt{C} \\ \forall i \in \{1, \ldots, n\}\ \Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_i^s \leadsto \mathtt{E}_i^b : \mathtt{T}_i\ \textbf{throws: } \mathtt{EE}_i \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\prec \mathit{Mth}(\mathtt{C}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n) = [\textsc{mk}{=}\mathtt{not\text{-}static}, \textsc{ret}{=}\mathtt{T}, \textsc{par}{=}\bar{\mathtt{T}}]\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash & \mathtt{E}_0^s.\mathtt{m}(\mathtt{E}_1^s, \ldots, \mathtt{E}_n^s) \leadsto \\ & \mathtt{E}_0^b. \ll \mathtt{C.T}\ (\bar{\mathtt{T}}) \gg_{\mathtt{vrt}} \mathtt{m}(\mathtt{E}_1^b, \ldots, \mathtt{E}_n^b) : \mathtt{T} \\ & \quad \textbf{throws: } \mathtt{EE}_0 \cup_{\mathtt{Exc}} \ldots \cup_{\mathtt{Exc}} \mathtt{EE}_n \cup_{\mathtt{Exc}} \varepsilon(\mathtt{RT} \not\prec \mathtt{C.m}(\mathtt{T}_1 \ldots \mathtt{T}_n))\end{array}}$$

$$(9)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_0^s \leadsto \mathtt{E}_0^b : \mathtt{I}\ \textbf{throws: } \mathtt{EE}_0 \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\prec \exists_{\mathtt{ifc}}\, \mathtt{I} \\ \forall i \in \{1, \ldots, n\}\ \Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_i^s \leadsto \mathtt{E}_i^b : \mathtt{T}_i\ \textbf{throws: } \mathtt{EE}_i \\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\prec \mathit{Mth}(\mathtt{I}, \mathtt{m}, \mathtt{T}_1 \ldots \mathtt{T}_n) = [\textsc{mk}{=}\mathtt{abstract}, \textsc{ret}{=}\mathtt{T}, \textsc{par}{=}\bar{\mathtt{T}}]\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash & \mathtt{E}_0^s.\mathtt{m}(\mathtt{E}_1^s, \ldots, \mathtt{E}_n^s) \leadsto \\ & \mathtt{E}_0^b. \ll \mathtt{I.T}\ (\bar{\mathtt{T}}) \gg_{\mathtt{int}} \mathtt{m}(\mathtt{E}_1^b, \ldots, \mathtt{E}_n^b) : \mathtt{T} \\ & \quad \textbf{throws: } \mathtt{EE}_0 \cup_{\mathtt{Exc}} \ldots \cup_{\mathtt{Exc}} \mathtt{EE}_n \cup_{\mathtt{Exc}} \varepsilon(\mathtt{RT} \not\prec \mathtt{I.m}(\mathtt{T}_1 \ldots \mathtt{T}_n))\end{array}}$$

Figure 4.15: Expression typing rules (1/3)

metarule (5). Because `this` can be used only in non-static contexts, the local type environment Π is defined on `this` only when it can be used; see Figure 4.20 and Figure 4.21.

A parameter name `x` is compiled into itself, has type $\Pi(\mathtt{x})$, and throws no exceptions, metarule (6).

An instance creation expression of a class `C` has type `C` and throws any exception that can be thrown by evaluating the arguments and the exceptions specified by the most specific constructor, metarule (7). The premise $\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists_{\mathtt{cls}} \epsilon\ \mathtt{C}$ ensures that the class `C` is both accessible from `RT` and not abstract.

Metarules from (8) to (11) define the compilation of a method invocation; for each invocation only one metarule can be instantiated, and each one compiles to a different binary expression. The first of them, metarule (8), models the compilation of a virtual method invocation. That is, an invocation of an instance method of a class `C`. The premise $\Gamma^{\mathrm{S}} \vdash \mathtt{RT} \not\lessdot \exists_{\mathtt{cls}}\ \mathtt{any}\ \mathtt{C}$ ensures that the type of the receiver $\mathtt{E}_0^s$ is a class `C`, which is accessible from `RT`. The whole expression has type `T`, the return type of the most specific method found, and throws any exception that can be thrown by evaluating the receiver $\mathtt{E}_0^s$ or the arguments $\mathtt{E}_i^s$, and the exceptions declared by the most specific method for the invocation.

The same reasoning applies to metarule (9), except that in this case the receiver must be an accessible interface `I` (so, the method kind of the most specific method must be `abstract`).

Metarule (10) deals with static invocations; in this case the receiver must be a class `C` (interfaces can only define abstract methods).

Finally, metarule (11) defines the compilation of invocations via keyword `super`. The invocation is resolved as any instance method invocation, but starting the search from the superclass $\mathtt{C}'$. Because $\mathtt{C}'$ is found using $\Pi(\mathtt{this})$, invocation via `super` cannot be compiled in static contexts, as it must be.

Metarules (12) to (14) define the compilation of the three kinds of field accesses: via `super`, instance and static access. Note that, differently from method invocations, a static field can be defined by both classes and interfaces. If the field is accessed via an interface type, then the field must necessarily be `final`, but we do not need to check this here because we assume Γ^{S} to be well-formed, hence any field found starting the search from an interface type is automatically `final`.

Metarules (15) to (18) define the compilation of the various forms of assignment, depending on the target. The target can be, respectively, a parameter, a field inherited by superclass (via `super`), an instance field and a static field. In all field assignments the type of the target is always a class `C`, because interface fields are always `static final` hence cannot be assigned (they can only be initialized once and for all via the initialization expression specified in the declaration).

$$
(10)\quad \frac{\begin{array}{l}\forall i \in \{1,\dots,n\}\ \Gamma^{\mathrm{s}};\Pi;\mathtt{RT} \vdash \mathtt{E}_i^s \rightsquigarrow \mathtt{E}_i^b : \mathtt{T}_i \ \textbf{throws: } \mathtt{EE}_i \\ \Gamma^{\mathrm{s}} \vdash \mathtt{RT} \not\prec \mathit{Mth}(\mathtt{C},\mathtt{m},\mathtt{T}_1\dots\mathtt{T}_n) = [\textsc{mk}{=}\mathtt{static}, \textsc{ret}{=}\mathtt{T}, \textsc{par}{=}\bar{\mathtt{T}}]\end{array}}{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT} \vdash \ \mathtt{C.m}(\mathtt{E}_1^s,\dots,\mathtt{E}_n^s) \rightsquigarrow \\ \qquad\qquad \ll \mathtt{C.T}\ (\bar{\mathtt{T}}) \gg_{\mathtt{stt}} \mathtt{m}(\mathtt{E}_1^b,\dots,\mathtt{E}_n^b) : \mathtt{T} \\ \qquad\qquad\quad \textbf{throws: } \mathtt{EE}_1 \cup_{\mathtt{Exc}} \dots \cup_{\mathtt{Exc}} \mathtt{EE}_n \cup_{\mathtt{Exc}} \varepsilon(\mathtt{RT} \not\prec \mathtt{C.m}(\mathtt{T}_1\dots\mathtt{T}_n))\end{array}}
$$

$$
(11)\quad \frac{\begin{array}{l}\forall i \in \{1,\dots,n\}\ \Gamma^{\mathrm{s}};\Pi;\mathtt{C} \vdash \mathtt{E}_i^s \rightsquigarrow \mathtt{E}_i^b : \mathtt{T}_i \ \textbf{throws: } \mathtt{EE}_i \\ \Gamma^{\mathrm{s}} \vdash \mathtt{C} \not\prec \mathit{Mth}(\mathtt{C}',\mathtt{m},\mathtt{T}_1\dots\mathtt{T}_n) = [\textsc{mk}{=}\mathtt{not\text{-}static}, \textsc{ret}{=}\mathtt{T}, \textsc{par}{=}\bar{\mathtt{T}}]\end{array}}{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{C} \vdash \ \mathtt{super.m}(\mathtt{E}_1^s,\dots,\mathtt{E}_n^s) \rightsquigarrow \\ \qquad\qquad \ll \mathtt{C'.T}\ (\bar{\mathtt{T}}) \gg_{\mathtt{spr}} \mathtt{m}(\mathtt{E}_1^b,\dots,\mathtt{E}_n^b) : \mathtt{T} \\ \qquad\qquad\quad \textbf{throws: } \mathtt{EE}_1 \cup_{\mathtt{Exc}} \dots \cup_{\mathtt{Exc}} \mathtt{EE}_n \cup_{\mathtt{Exc}} \varepsilon(\mathtt{C} \not\prec \mathtt{C'.m}(\mathtt{T}_1\dots\mathtt{T}_n))\end{array}} \qquad \begin{array}{l}\Pi(\mathtt{this}) = \mathtt{C} \\ \Gamma^{\mathrm{s}}(\mathtt{C}).\textsc{parent} = \mathtt{C}'\end{array}
$$

$$
(12)\quad \frac{\Gamma^{\mathrm{s}} \vdash \mathtt{C} \not\prec \mathit{Fld}(\mathtt{C}',\mathtt{f}) = [\textsc{final}{=}\mathtt{any}, \textsc{fk}{=}\epsilon, \textsc{t}{=}\mathtt{T}]}{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{C} \vdash \ \mathtt{super.f} \rightsquigarrow \\ \qquad\qquad \mathtt{this}. \ll \mathtt{C'.T} \gg_{\mathtt{if}} \mathtt{f} : \mathtt{T} \\ \qquad\qquad\quad \textbf{throws: } \emptyset\end{array}} \qquad \begin{array}{l}\Pi(\mathtt{this}) = \mathtt{C} \\ \Gamma^{\mathrm{s}}(\mathtt{C}).\textsc{parent} = \mathtt{C}'\end{array}
$$

$$
(13)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT} \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{C} \ \textbf{throws: } \mathtt{EE} \\ \Gamma^{\mathrm{s}} \vdash \mathtt{RT} \not\prec \mathit{Fld}(\mathtt{C},\mathtt{f}) = [\textsc{final}{=}\mathtt{any}, \textsc{fk}{=}\epsilon, \textsc{t}{=}\mathtt{T}]\end{array}}{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT} \vdash \ \mathtt{E}^s.\mathtt{f} \rightsquigarrow \\ \qquad\qquad \mathtt{E}^b. \ll \mathtt{C.T} \gg_{\mathtt{if}} \mathtt{f} : \mathtt{T} \\ \qquad\qquad\quad \textbf{throws: } \mathtt{EE}\end{array}}
$$

$$
(14)\quad \frac{\Gamma^{\mathrm{s}} \vdash \mathtt{RT} \not\prec \mathit{Fld}(\mathtt{RT}',\mathtt{f}) = [\textsc{final}{=}\mathtt{any}, \textsc{fk}{=}\mathtt{static}, \textsc{t}{=}\mathtt{T}]}{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT} \vdash \ \mathtt{RT'.f} \rightsquigarrow \\ \qquad\qquad \ll \mathtt{RT'.T} \gg_{\mathtt{sf}} \mathtt{f} : \mathtt{T} \\ \qquad\qquad\quad \textbf{throws: } \emptyset\end{array}}
$$

$$
(15)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT} \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{T} \ \textbf{throws: } \mathtt{EE} \\ \Gamma^{\mathrm{s}} \vdash \mathtt{T} \leq \mathtt{T}'\end{array}}{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT} \vdash \ \mathtt{x} = \mathtt{E}^s \rightsquigarrow \\ \qquad\qquad \mathtt{x} = \mathtt{E}^b : \mathtt{T}' \\ \qquad\qquad\quad \textbf{throws: } \mathtt{EE}\end{array}} \qquad \Pi(\mathtt{x}) = \mathtt{T}'
$$

Figure 4.16: Expression typing rules (2/3)

$$
(16)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{s}}(\mathtt{C}).\textsc{parent} = \mathtt{C}' \\ \Gamma^{\mathrm{s}} \vdash \mathtt{C} \not\prec \mathit{Fld}(\mathtt{C}', \mathtt{f}) = [\textsc{final}{=}\epsilon, \textsc{fk}{=}\epsilon, \textsc{t}{=}\mathtt{T}] \\ \Gamma^{\mathrm{s}}; \Pi; \mathtt{C} \vdash \mathtt{E}^{s} \rightsquigarrow \mathtt{E}^{b} : \mathtt{T}' \textbf{ throws: } \mathtt{EE} \\ \Gamma^{\mathrm{s}} \vdash \mathtt{T}' \leq \mathtt{T}\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}}; \Pi; \mathtt{C} \vdash & \mathtt{super.f} = \mathtt{E}^{s} \rightsquigarrow \\ & \mathtt{this.} \ll \mathtt{C}'.\mathtt{T} \gg_{\mathtt{if}} \mathtt{f} = \mathtt{E}^{b} : \mathtt{T} \\ & \quad \textbf{throws: } \mathtt{EE}\end{array}} \qquad \Pi(\mathtt{this}) = \mathtt{C}
$$

$$
(17)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{s}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_1^{s} \rightsquigarrow \mathtt{E}_1^{b} : \mathtt{C} \textbf{ throws: } \mathtt{EE}_1 \\ \Gamma^{\mathrm{s}} \vdash \mathtt{RT} \not\prec \mathit{Fld}(\mathtt{C}, \mathtt{f}) = [\textsc{final}{=}\epsilon, \textsc{fk}{=}\epsilon, \textsc{t}{=}\mathtt{T}] \\ \Gamma^{\mathrm{s}}; \Pi; \mathtt{RT} \vdash \mathtt{E}_2^{s} \rightsquigarrow \mathtt{E}_2^{b} : \mathtt{T}' \textbf{ throws: } \mathtt{EE}_2 \\ \Gamma^{\mathrm{s}} \vdash \mathtt{T}' \leq \mathtt{T}\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}}; \Pi; \mathtt{RT} \vdash & \mathtt{E}_1^{s}.\mathtt{f} = \mathtt{E}_2^{s} \rightsquigarrow \\ & \mathtt{E}_1^{b}. \ll \mathtt{C.T} \gg_{\mathtt{if}} \mathtt{f} = \mathtt{E}_2^{b} : \mathtt{T} \\ & \quad \textbf{throws: } \mathtt{EE}_1 \cup_{\mathtt{Exc}} \mathtt{EE}_2\end{array}}
$$

$$
(18)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{s}} \vdash \mathtt{RT} \not\prec \mathit{Fld}(\mathtt{C}, \mathtt{f}) = [\textsc{final}{=}\epsilon, \textsc{fk}{=}\mathtt{static}, \textsc{t}{=}\mathtt{T}] \\ \Gamma^{\mathrm{s}}; \Pi; \mathtt{RT} \vdash \mathtt{E}^{s} \rightsquigarrow \mathtt{E}^{b} : \mathtt{T}' \textbf{ throws: } \mathtt{EE} \\ \Gamma^{\mathrm{s}} \vdash \mathtt{T}' \leq \mathtt{T}\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}}; \Pi; \mathtt{RT} \vdash & \mathtt{C.f} = \mathtt{E}^{s} \rightsquigarrow \\ & \ll \mathtt{C.T} \gg_{\mathtt{sf}} \mathtt{f} = \mathtt{E}^{b} : \mathtt{T} \\ & \quad \textbf{throws: } \mathtt{EE}\end{array}}
$$

Figure 4.17: Expression typing rules (3/3)

Compilation of statements is expressed by the following judgment:

$$\Gamma^{\mathrm{S}}; \Pi; \mathtt{RT} \vdash \mathtt{STMT}^s \rightsquigarrow \mathtt{STMT}^b \ \textbf{throws: } \mathtt{EE}\ \textbf{ccn: } \beta$$

with the meaning "statement $\mathtt{STMT}^s$ is compiled to $\mathtt{STMT}^b$, throws exceptions `EE` and can complete normally (boolean flag β) when contained in type `RT`, in a local environment Π, and in a type environment Γ^{S}". Figure 4.18 and Figure 4.19 show the metarules defining this judgment.

Both the exception expression `EE` and the flag β are required to model unreachable code. The intuitive idea is that a reachable statement can complete normally (so the flag β is true) when, at run-time, the flow of execution may continue beyond such a statement. For instance, a sequence of statements can complete normally if and only if all statements it consists of can complete normally (see metarules (1) and (2) of Figure 4.18), the evaluation of a statement expression $\mathtt{SE}^b$ always completes normally (metarule (3)), while a statement `break` cannot ever complete normally (metarule (4)). These notions are described in detail in JLS 14.20. The metarules contain an apparent asymmetry in how the statements `if` (metarule (5)) and `while` (metarule (7)) are handled. That is, `while (false)` {...} is not allowed, because its body would be unreachable, but the similar cases `if (false)` {...} `else` {...} and `if (true)` {...} `else` {...} are ok (even though in both cases one of the two branches is indeed unreachable). The typing rules reflect the language specification which allows this use of statement `if` as a way to express conditional compilation.

The two metarules in Figure 4.19 deal with throwing and catching exceptions. A statement `throw`, whose expression $\mathtt{E}^s$ has type `C`, can throw all the exceptions the evaluation of $\mathtt{E}^s$ can throw, plus, of course, `C` itself. A statement `throw` never completes normally. The last rule is the trickiest of all: a statement `try` can throw the exceptions its body can throw $\mathtt{EE}_0$, minus all the caught ones $\{\mathtt{C}_1, \ldots \mathtt{C}_n\}$, plus the ones that can be thrown by statement inside its `catch` or `finally` clauses. Moreover, the statement `try` can complete normally only when at least its body $\mathtt{STMT}_0^s$ or one of the catch clauses $\mathtt{STMT}_i^s$ can complete normally *and* the `finally` clause can complete normally.

In our model "running the compilation" on a fragment `S` amounts to derive the following judgment:

$$\Gamma^{\mathrm{S}} \vdash \mathtt{S} \rightsquigarrow \mathtt{B}$$

Figure 4.20 and Figure 4.21 show the metarules defining this judgment.

In defining the compilation of a set of classes we assume to compile them one by one, that is, we assume that no global optimizations take place. This reflects the fact that in languages with dynamic linking like Java, the concept of "program" is only significant at run-time so it is safer to leave cross class optimizations to virtual machines like HotSpot [SUN01].

$$(1)\quad \frac{}{\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \{\}\leadsto\{\}\ \textbf{throws: }\emptyset\ \textbf{ccn: }\mathtt{true}}$$

$$(2)\quad \frac{\begin{array}{l} i\in 1..(n-1)\ \Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{STMT}^s_i\leadsto\mathtt{STMT}^b_i\ \textbf{throws: }\mathtt{EE}_i\ \textbf{ccn: }\mathtt{true}\\ \Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{STMT}^s_n\leadsto\mathtt{STMT}^b_n\ \textbf{throws: }\mathtt{EE}_n\ \textbf{ccn: }\beta\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash & \{\mathtt{STMT}^s_1\ldots\mathtt{STMT}^s_n\}\leadsto\\ & \{\mathtt{STMT}^b_1\ldots\mathtt{STMT}^b_n\}\\ & \ \textbf{throws: }\mathtt{EE}_1\cup_{\mathtt{Exc}}\ldots\cup_{\mathtt{Exc}}\mathtt{EE}_n\ \textbf{ccn: }\beta\end{array}}\qquad n\geq 1$$

$$(3)\quad \frac{\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{SE}^s\leadsto\mathtt{SE}^b:\mathtt{T}\ \textbf{throws: }\mathtt{EE}}{\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{SE}^s\ ;\leadsto\mathtt{SE}^b\ ;\ \textbf{throws: }\mathtt{EE}\ \textbf{ccn: }\mathtt{true}}$$

$$(4)\quad \frac{}{\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{break}\ ;\leadsto\mathtt{break}\ ;\ \textbf{throws: }\emptyset\ \textbf{ccn: }\mathtt{false}}$$

$$(5)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{E}^s\leadsto\mathtt{E}^b:\mathtt{bool}\ \textbf{throws: }\mathtt{EE}_0\\ i\in 1..2\ \Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{STMT}^s_i\leadsto\mathtt{STMT}^b_i\ \textbf{throws: }\mathtt{EE}_i\ \textbf{ccn: }\beta_i\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash & \mathtt{if}\ (\mathtt{E}^s)\ \mathtt{STMT}^s_1\ \mathtt{else}\ \mathtt{STMT}^s_2\leadsto\\ & \mathtt{if}\ (\mathtt{E}^b)\ \mathtt{STMT}^b_1\ \mathtt{else}\ \mathtt{STMT}^b_2\\ & \ \textbf{throws: }\mathtt{EE}_0\cup_{\mathtt{Exc}}\mathtt{EE}_1\cup_{\mathtt{Exc}}\mathtt{EE}_2\ \textbf{ccn: }\beta_1\vee\beta_2\end{array}}$$

$$(6)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{E}^s\leadsto\mathtt{E}^b:\mathtt{bool}\ \textbf{throws: }\mathtt{EE}_0\\ \Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{STMT}^s\leadsto\mathtt{STMT}^b\ \textbf{throws: }\mathtt{EE}_1\ \textbf{ccn: }\_\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash & \mathtt{while}\ (\mathtt{E}^s)\ \mathtt{STMT}^s\leadsto\\ & \mathtt{while}\ (\mathtt{E}^b)\ \mathtt{STMT}^b\\ & \ \textbf{throws: }\mathtt{EE}_0\cup_{\mathtt{Exc}}\mathtt{EE}_1\ \textbf{ccn: }\mathtt{true}\end{array}}\qquad \mathtt{E}^b\text{ is not a value}$$

$$(7)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{E}^s\leadsto\mathtt{true}:\mathtt{bool}\ \textbf{throws: }\emptyset\\ \Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{STMT}^s\leadsto\mathtt{STMT}^b\ \textbf{throws: }\mathtt{EE}\ \textbf{ccn: }\_\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash & \mathtt{while}\ (\mathtt{E}^s)\ \mathtt{STMT}^s\leadsto\\ & \mathtt{while}\ (\mathtt{true})\ \mathtt{STMT}^b\\ & \ \textbf{throws: }\mathtt{EE}\ \textbf{ccn: }\beta\end{array}}\qquad \beta=\begin{array}{l}\text{there is a break for this}\\ \mathtt{while}\text{ inside }\mathtt{STMT}^s\end{array}$$

Figure 4.18: Statement typing rules (1/2)

$$(8)\quad \frac{\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{E}^s \leadsto \mathtt{E}^b : \mathtt{C}\ \textbf{throws: }\mathtt{EE}}{\begin{array}{ll}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash & \mathtt{throw\ E}^s\ \mathtt{;} \leadsto \\ & \mathtt{throw\ E}^b\ \mathtt{;} \\ & \textbf{throws: }\mathtt{EE}\cup_{\mathtt{Exc}}\{\mathtt{C}\}\ \textbf{ccn: }\mathtt{false}\end{array}}$$

$$(9)\quad \frac{\begin{array}{l}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{STMT}^s_0 \leadsto \mathtt{STMT}^b_0\ \textbf{throws: }\mathtt{EE}_0\ \textbf{ccn: }\beta_0 \\ i\in 1..n\ \Gamma^{\mathrm{s}};\Pi[\mathtt{x}_i\mapsto \mathtt{C}_i];\mathtt{RT}\vdash \mathtt{STMT}^s_i \leadsto \mathtt{STMT}^b_i\ \textbf{throws: }\mathtt{EE}_i\ \textbf{ccn: }\beta_i \\ \Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash \mathtt{STMT}^s_{n+1} \leadsto \mathtt{STMT}^b_{n+1}\ \textbf{throws: }\mathtt{EE}_{n+1}\ \textbf{ccn: }\beta_{n+1} \\ i\in 1..n\ \Gamma^{\mathrm{s}}\vdash \mathtt{RT}\not\leqslant (\mathtt{EE}_0\setminus_{\mathtt{Exc}}\{\mathtt{C}_1,\ldots,\mathtt{C}_{i-1}\})\supseteq_{\mathtt{Exc}}\{\mathtt{C}_i\}\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}};\Pi;\mathtt{RT}\vdash & \mathtt{try}\ \{\ \mathtt{STMT}^s_0\ \} \\ & \mathtt{catch}\ (\mathtt{C}_1\ \mathtt{x}_1)\ \{\ \mathtt{STMT}^s_1\ \} \\ & \ldots \\ & \mathtt{catch}\ (\mathtt{C}_n\ \mathtt{x}_n)\ \{\ \mathtt{STMT}^s_n\ \} \\ & \mathtt{finally}\ \{\ \mathtt{STMT}^s_{n+1}\ \}\ \leadsto \\ & \mathtt{try}\ \{\ \mathtt{STMT}^b_0\ \} \\ & \mathtt{catch}\ (\mathtt{C}_1\ \mathtt{x}_1)\ \{\ \mathtt{STMT}^b_1\ \} \\ & \ldots \\ & \mathtt{catch}\ (\mathtt{C}_n\ \mathtt{x}_n)\ \{\ \mathtt{STMT}^b_n\ \} \\ & \mathtt{finally}\ \{\ \mathtt{STMT}^b_{n+1}\ \} \\ & \textbf{throws: }(\mathtt{EE}_0\setminus_{\mathtt{Exc}}\{\mathtt{C}_1,\ldots,\mathtt{C}_n\})\cup_{\mathtt{Exc}}\mathtt{EE}_1\cup_{\mathtt{Exc}}\ldots\cup_{\mathtt{Exc}}\mathtt{EE}_{n+1} \\ & \textbf{ccn: }(\beta_0\vee\ldots\vee\beta_n)\wedge\beta_{n+1}\end{array}}$$

Figure 4.19: Statement typing rules (2/2)

$$
(1)\ \frac{\begin{array}{l}\vdash \Gamma^{\mathrm{s}}\diamond \\ \Gamma^{\mathrm{s}};\mathtt{C} \vdash \mathtt{KDS}^{s} \leadsto \mathtt{KDS}^{b} \\ \Gamma^{\mathrm{s}};\mathtt{C} \vdash \mathtt{FDS}^{s} \leadsto \mathtt{FDS}^{b} \\ \Gamma^{\mathrm{s}};\mathtt{C} \vdash \mathtt{MDS}^{s} \leadsto \mathtt{MDS}^{b}\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}} \vdash & \mathtt{S} \leadsto \\ & \mathtt{AM\ CK\ class\ C\ extends\ C'} \\ & \quad \mathtt{implements\ I}_1,\ldots,\mathtt{I}_m\ \{ \\ & \quad \mathtt{KDS}^{b} \\ & \quad \mathtt{FDS}^{b} \\ & \quad \mathtt{MDS}^{b} \\ & \}\end{array}}
\qquad
\begin{array}{ll}\mathtt{S} = & \mathtt{AM\ CK\ class\ C\ extends\ C'} \\ & \quad \mathtt{implements\ I}_1,\ldots,\mathtt{I}_m\ \{ \\ & \quad \mathtt{KDS}^{s} \\ & \quad \mathtt{FDS}^{s} \\ & \quad \mathtt{MDS}^{s} \\ & \} \\ \multicolumn{2}{l}{\mathcal{T}(\mathtt{S}) = \Gamma^{\mathrm{s}}(\mathtt{C})}\end{array}
$$

$$
(2)\ \frac{\begin{array}{l}\vdash \Gamma^{\mathrm{s}}\diamond \\ \Gamma^{\mathrm{s}};\mathtt{I} \vdash \mathtt{FDS}^{s} \leadsto \mathtt{FDS}^{b} \\ \Gamma^{\mathrm{s}};\mathtt{I} \vdash \mathtt{MDS}^{s} \leadsto \mathtt{MDS}^{b}\end{array}}{\begin{array}{ll}\Gamma^{\mathrm{s}} \vdash & \mathtt{S} \leadsto \\ & \mathtt{AM\ interface\ I\ extends\ I}_1,\ldots,\mathtt{I}_m\ \{ \\ & \quad \mathtt{FDS}^{b} \\ & \quad \mathtt{MDS}^{b} \\ & \}\end{array}}
\qquad
\begin{array}{ll}\mathtt{S} = & \mathtt{AM\ interface\ I} \\ & \quad \mathtt{extends\ I}_1,\ldots,\mathtt{I}_m\ \{ \\ & \quad \mathtt{FDS}^{s} \\ & \quad \mathtt{MDS}^{s} \\ & \} \\ \multicolumn{2}{l}{\mathcal{T}(\mathtt{S}) = \Gamma^{\mathrm{s}}(\mathtt{I})}\end{array}
$$

$$
(3)\ \frac{\forall i \in 1..n\ \Gamma^{\mathrm{s}};\mathtt{C} \vdash \mathtt{KD}_i^{s} \leadsto \mathtt{KD}_i^{b}}{\Gamma^{\mathrm{s}};\mathtt{C} \vdash \mathtt{KD}_1^{s}\ldots\mathtt{KD}_n^{s} \leadsto \mathtt{KD}_1^{b}\ldots\mathtt{KD}_n^{b}}
$$

$$
(4)\ \frac{\forall i \in 1..n\ \Gamma^{\mathrm{s}};\mathtt{RT} \vdash \mathtt{FD}_i^{s} \leadsto \mathtt{FD}_i^{b}}{\Gamma^{\mathrm{s}};\mathtt{RT} \vdash \mathtt{FD}_1^{s}\ldots\mathtt{FD}_n^{s} \leadsto \mathtt{FD}_1^{b}\ldots\mathtt{FD}_n^{b}}
$$

$$
(5)\ \frac{\forall i \in 1..n\ \Gamma^{\mathrm{s}};\mathtt{RT} \vdash \mathtt{MD}_i^{s} \leadsto \mathtt{MD}_i^{b}}{\Gamma^{\mathrm{s}};\mathtt{RT} \vdash \mathtt{MD}_1^{s}\ldots\mathtt{MD}_n^{s} \leadsto \mathtt{MD}_1^{b}\ldots\mathtt{MD}_n^{b}}
$$

Figure 4.20: Compilation rules (1/2)

$$
(6)\ \frac{\begin{array}{l}
\forall i \in \{1,\dots,n\}\ \Gamma^{\mathrm{s}} \vdash \mathtt{C} \not\prec \exists\, \mathtt{T}_i \\
\forall i \in \{1,\dots,m\}\ \Gamma^{\mathrm{s}};\Pi;\mathtt{C} \vdash \mathtt{E}^s_i \rightsquigarrow \mathtt{E}^b_i : \mathtt{T}'_i\ \textbf{throws: } \mathtt{EE}_i \\
\Gamma^{\mathrm{s}};\Pi;\mathtt{C} \vdash \mathtt{STMTS}^s \rightsquigarrow \mathtt{STMTS}^b\ \textbf{throws: } \mathtt{EE}_0\ \textbf{ccn: } \_ \\
\Gamma^{\mathrm{s}} \vdash \mathtt{C} \not\prec \mathit{Cns}(\mathtt{C}', \mathtt{T}'_1 \dots \mathtt{T}'_m) = [_{\text{PAR}=}\bar{\mathtt{T}}] \\
\Gamma^{\mathrm{s}} \vdash \mathtt{C} \not\prec (\mathtt{EE}_0 \cup_{\mathtt{Exc}} \dots \cup_{\mathtt{Exc}} \mathtt{ES}_n \cup_{\mathtt{Exc}} \varepsilon(\mathtt{C} \not\prec \mathtt{C}'(\bar{\mathtt{T}}))) \subseteq_{\mathtt{Exc}} \mathtt{ES}
\end{array}}{\begin{array}{ll}
\Gamma^{\mathrm{s}};\mathtt{C} \vdash & \mathtt{AM}\ (\mathtt{T}_1\ \mathtt{x}_1, \dots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws}\ \mathtt{ES}\ \{ \\
& \quad \mathtt{super}(\mathtt{E}^s_1, \dots, \mathtt{E}^s_m);\ \mathtt{STMTS}^s \\
& \}\ \rightsquigarrow \\
& \mathtt{AM}\ (\mathtt{T}_1\ \mathtt{x}_1, \dots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws}\ \mathtt{ES}\ \{ \\
& \quad \mathtt{super}(\mathtt{E}^b_1, \dots, \mathtt{E}^b_n) \ll \mathtt{C}'(\bar{\mathtt{T}}) \gg_{\mathtt{c}};\ \mathtt{STMTS}^b \\
& \}
\end{array}}
\qquad
\begin{array}{l}
\Gamma^{\mathrm{s}}(\mathtt{C}).\text{PARENT} = \mathtt{C}' \\
\Pi = \{\ x_1 \mapsto \mathtt{T}_1, \\
\qquad \dots, \\
\qquad \mathtt{x}_n \mapsto \mathtt{T}_n, \\
\qquad \mathtt{this} \mapsto \mathtt{C}\}
\end{array}
$$

$$
(7)\ \frac{\begin{array}{l}
\Gamma^{\mathrm{s}};\Pi;\mathtt{RT} \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{T}'\ \textbf{throws: } \mathtt{EE} \\
\Gamma^{\mathrm{s}} \vdash \mathtt{RT} \not\prec \mathtt{EE} \subseteq_{\mathtt{Exc}} \emptyset \\
\Gamma^{\mathrm{s}} \vdash \mathtt{T}' \leq \mathtt{T}
\end{array}}{\Gamma^{\mathrm{s}};\mathtt{RT} \vdash \mathtt{AM\ FINAL\ FK\ T\ f} = \mathtt{E}^s\ ;\ \rightsquigarrow \mathtt{AM\ FINAL\ FK\ T\ f} = \mathtt{E}^b\ ;}
\qquad
\Pi = \mathit{This}(\mathtt{FK}, \mathtt{C})
$$

$$
(8)\ \frac{\begin{array}{l}
\forall i \in \{0,\dots,n\}\ \Gamma^{\mathrm{s}} \vdash \mathtt{C} \not\prec \exists\, \mathtt{T}_i \\
\Gamma^{\mathrm{s}};\Pi;\mathtt{C} \vdash \mathtt{STMTS}^s \rightsquigarrow \mathtt{STMTS}^b\ \textbf{throws: } \mathtt{EE}\ \textbf{ccn: } \mathtt{true} \\
\Gamma^{\mathrm{s}};\Pi;\mathtt{C} \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{T}'\ \textbf{throws: } \mathtt{EE}' \\
\Gamma^{\mathrm{s}} \vdash \mathtt{C} \not\prec \mathtt{EE} \cup_{\mathtt{Exc}} \mathtt{EE}' \subseteq_{\mathtt{Exc}} \mathtt{ES} \\
\Gamma^{\mathrm{s}} \vdash \mathtt{T}' \leq \mathtt{T}_0
\end{array}}{\begin{array}{ll}
\Gamma^{\mathrm{s}};\mathtt{C} \vdash & \mathtt{AM\ MK\ T_0\ m}(\mathtt{T}_1\ \mathtt{x}_1, \dots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws}\ \mathtt{ES}\ \{ \\
& \quad \mathtt{STMTS}^s\ \mathtt{return}\ \mathtt{E}^s; \\
& \} \rightsquigarrow \\
& \mathtt{AM\ MK\ T_0\ m}(\mathtt{T}_1\ \mathtt{x}_1, \dots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws}\ \mathtt{ES}\ \{ \\
& \quad \mathtt{STMTS}^b\ \mathtt{return}\ \mathtt{E}^b; \\
& \}
\end{array}}
\qquad
\begin{array}{l}
\Pi = \{\ \mathtt{x}_1 \mapsto \mathtt{T}_1, \\
\qquad \dots, \\
\qquad \mathtt{x}_n \mapsto \mathtt{T}_n\} \cup \\
\qquad \mathit{This}(\mathtt{MK}, \mathtt{C}) \\
\mathtt{MK} \neq \mathtt{abstract}
\end{array}
$$

$$
(9)\ \frac{\forall i \in \{0,\dots,n\}\ \Gamma^{\mathrm{s}} \vdash \mathtt{RT} \not\prec \exists\, \mathtt{T}_i}{\begin{array}{ll}
\Gamma^{\mathrm{s}};\mathtt{RT} \vdash & \mathtt{AM\ abstract\ T_0\ m}(\mathtt{T}_1\ \mathtt{x}_1, \dots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws\ ES} \rightsquigarrow \\
& \mathtt{AM\ abstract\ T_0\ m}(\mathtt{T}_1\ \mathtt{x}_1, \dots, \mathtt{T}_n\ \mathtt{x}_n)\ \mathtt{throws\ ES}
\end{array}}
$$

$$
\begin{array}{l}
\mathit{This}(\_, \mathtt{I}) = \emptyset \\
\mathit{This}(\mathtt{static}, \mathtt{C}) = \emptyset; \\
\mathit{This}(\epsilon, \mathtt{C}) = \{\mathtt{this} \mapsto \mathtt{C}\}
\end{array}
$$

Figure 4.21: Compilation rules (2/2)

4.2.5 Proofs

As we stated in Section 3.3, in this system the set of type assumptions Γ used in the proof tree of a compilation judgment

$$\Gamma_{\mathtt{old}} \vdash \mathtt{S} \leadsto \mathtt{B}$$

are the *weakest* type assumptions Γ^{NS} needed for compiling `S` to `B`.

That is, as long as a new environment entails Γ, there is no need to recompile the source as its recompilation would produce the same binary.

Theorem 4.2.9 proves this important property, using the following auxiliary lemmas and theorems.

Lemma 4.2.1 *If*

- *judgments* $\_;\Pi;RT \vdash \_ \leadsto \mathtt{E}^b : T_1$ ***throws:*** $\_$ *and*
- $\_;\Pi;RT \vdash \_ \leadsto \mathtt{E}^b : T_2$ ***throws:*** $\_$ *are derivable,*

then $T_1 = T_2$.

Proof Trivial due to the annotations of constructor/method invocations: if a source expression $\mathtt{E}^s$ is successfully compiled into a binary expression $\mathtt{E}^b$, then the type of $\mathtt{E}^s$ can be directly extracted from Π and $\mathtt{E}^b$.

□

Lemma 4.2.2 *If*

- *judgments* $\_;\Pi;RT \vdash \_ \leadsto \mathtt{E}^b : \_$ ***throws:*** EE_1 *and*
- $\_;\Pi;RT \vdash \_ \leadsto \mathtt{E}^b : \_$ ***throws:*** EE_2 *are derivable,*

then $EE_1 = EE_2$.

Proof This can be proved by structural induction on the metarules defining the judgment. The proof is trivial on all the metarules except for:

- Metarule (7): the first part of the exception expression is the same by inductive hypothesis, the last part is completely determined by `RT` and $\mathtt{E}^b$;

- Metarule (8): the first part of the exception expression is the same by inductive hypothesis, the last part is determined by RT, $\mathtt{E}^b$ and Lemma 4.2.1;
- Metarules (9), (10) and (11): analogous to metarule (8).

□

Lemma 4.2.3 *If*

- *judgments* $\_; \Pi; RT \vdash \_ \rightsquigarrow \mathtt{E}^b : T_1$ ***throws:*** EE_1 *and*
- $\_; \Pi; RT \vdash \_ \rightsquigarrow \mathtt{E}^b : T_2$ ***throws:*** EE_2 *are derivable,*

then $T_1 = T_2$ *and* $EE_1 = EE_2$*.*

Proof Follows by Lemma 4.2.1 and Lemma 4.2.2.

□

Theorem 4.2.4 *If the judgment* $\Gamma_1^s; \Pi; RT \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : T$ ***throws:*** EE *can be derived, using the type assumptions* $\Gamma = \{\gamma_1, \ldots, \gamma_n\}$ *in the proof tree, then for all type environment* Γ_2^s *such that* $\Gamma_2^s(RT) = \Gamma_1^s(RT)$*:*

$$\Gamma_2^s; \Pi; RT \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \_ \ \textbf{\textit{throws:}}\ \_ \iff \Gamma_2^s \vdash \Gamma$$

Proof

$\Rightarrow$)

First, by Lemma 4.2.3, we know that any derivation $\Gamma_2^s; \Pi; \mathtt{RT} \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \_$ **throws:** _ must actually be a derivation for $\Gamma_2^s; \Pi; \mathtt{RT} \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{T}$ **throws:** EE.

Second, we note there is only one metarule to derive any binary expression, so the proof tree in Γ_2^s must consist of the same metarules of the one in Γ_1^s.

To prove that even the type assumptions used as premises of the metarules are the same, we note that every variable of these assumptions is:

- fixed by the metarule, or
- RT, or
- contained in $\mathtt{E}^s$ or in $\mathtt{E}^b$, or

- the type of a subexpression of $\mathtt{E}^s$, which is the same in every environment by Lemma 4.2.1.

For these reasons, to compile $\mathtt{E}^s$ in $\mathtt{E}^b$ inside RT there is no choice but using the same assumptions Γ in every type environment. Therefore, it must be $\Gamma^{\mathrm{s}}_2 \vdash \Gamma$ as well.

$\Leftarrow$)

Trivial: the proof tree of $\Gamma^{\mathrm{s}}_1; \Pi; \mathtt{RT} \vdash \mathtt{E}^s \rightsquigarrow \mathtt{E}^b : \mathtt{T}_1$ **throws:** $\mathtt{EE}_1$ can be derived in Γ^{s}_2 as well because the side-conditions only depend on Π and $\Gamma^{\mathrm{s}}_2(\mathtt{RT})$, which is equal to $\Gamma^{\mathrm{s}}_1(\mathtt{RT})$ by hypothesis.

□

Theorem 4.2.4 fixes three things: the name of the type RT containing the expression, the local type environment Π and the standard type environment for RT, that is, $\Gamma^{\mathrm{s}}(\mathtt{RT})$. On the one hand, if we think only in term of compiling a source expression $\mathtt{E}^s$ into a binary expression $\mathtt{E}^b$, these requirements are stronger than they could be: for instance, an unused parameter need not to be in Π. If a method receives three arguments but it does not use any of them, the result of the compilation of its body, and any contained expression, is necessarily independent from Π. So, we could avoid fixing RT, Π and $\Gamma^{\mathrm{s}}(\mathtt{RT})$ using some form of type assumptions, as we have done for the other requirements.

On the other hand, in Java expressions cannot be compiled in isolation. The only way to compile an expression $\mathtt{E}^s$ is to compile the fragment S which contains the declaration of the type RT containing $\mathtt{E}^s$. Because RT, Π and $\Gamma^{\mathrm{s}}(\mathtt{RT})$ are extracted directly from S, they can only differs from a previous version, used to compile $\mathtt{E}^s$, when the fragment S is changed. But, if S is changed, then it has to be recompiled anyway, so for our particular application it does not make sense trying to weaken the requirements for compiling expression any more.

Compilation of statements enjoys properties similar to the compilation of expressions, as the following lemmas and theorem state.

Lemma 4.2.5 *If*

- *judgments* $\_; \Pi; RT \vdash \_ \rightsquigarrow \mathtt{STMT}^b$ ***throws:*** EE_1 ***ccn:*** $\_$ *and*
- $\_; \Pi; RT \vdash \_ \rightsquigarrow \mathtt{STMT}^b$ ***throws:*** EE_2 ***ccn:*** $\_$ *are derivable,*

then $EE_1 = EE_2$.

Proof This can be proved by structural induction on the metarules defining the judgment. The proof is trivial on all the metarules except for:

- Metarule (3): follows by Lemma 4.2.2;
- Metarule (5): follows by inductive hypothesis and Lemma 4.2.2;
- Metarules(6) and (7): analogous to metarule (5);
- Metarule (8): follows by inductive hypothesis and Lemma 4.2.3;
- Metarule (9): follows by inductive hypothesis and the fact that the set $\{\mathtt{C}_1, \ldots, \mathtt{C}_n\}$ is determined by $\mathtt{E}^b$.

□

Lemma 4.2.6 *If*

- *judgments* $\_;\Pi; \mathit{RT} \vdash \_ \leadsto \mathtt{STMT}^b$ ***throws:*** $\_$ ***ccn:*** β_1 *and*
- $\_;\Pi; \mathit{RT} \vdash \_ \leadsto \mathtt{STMT}^b$ ***throws:*** $\_$ ***ccn:*** β_2 *are derivable,*

then $\beta_1 = \beta_2$.

Proof This can be proved by structural induction on the metarules defining the judgment. The proof is trivial on all the metarules except for metarule (7); in this case note that β is determined by $\mathtt{E}^b$.

□

Lemma 4.2.7 *If*

- *judgments* $\_;\Pi; \mathit{RT} \vdash \_ \leadsto \mathtt{STMT}^b$ ***throws:*** EE_1 ***ccn:*** β_1 *and*
- $\_;\Pi; \mathit{RT} \vdash \_ \leadsto \mathtt{STMT}^b$ ***throws:*** EE_2 ***ccn:*** β_2*,*

then $\mathit{EE}_1 = \mathit{EE}_2$ *and* $\beta_1 = \beta_2$.

Proof Follows by Lemma 4.2.5 and Lemma 4.2.6.

□

Theorem 4.2.8 *If the judgment* $\Gamma_1^s; \Pi; \mathit{RT} \vdash \mathtt{STMT}^s \leadsto \mathtt{STMT}^b$ ***throws:*** EE ***ccn:*** β *can be derived, using the type assumptions* $\Gamma = \{\gamma_1, \ldots, \gamma_n\}$ *in the proof tree, then for all type environment* Γ_2^s *such that* $\Gamma_2^s(\mathit{RT}) = \Gamma_1^s(\mathit{RT})$*:*

$$\Gamma_2^s; \Pi; \mathit{RT} \vdash \mathtt{STMT}^s \leadsto \mathtt{STMT}^b \textbf{ throws: } \_ \textbf{ ccn: } \_ \iff \Gamma_2^s \vdash \Gamma$$

Proof This theorem is analogous to Theorem 4.2.4, and the thesis can be proved with the same reasoning, using Lemma 4.2.7 and Theorem 4.2.4 for the expressions contained in the statement.

□

Building up the results shown so far, the following theorem shows that the type assumptions used in compiling a fragment `S` totally characterize its compilation into a particular binary `B`. This result is the key to obtain the sound and minimal compilation strategy we strive for.

Theorem 4.2.9 *If $\Gamma_{\texttt{old}}$ and $\Gamma_{\texttt{new}}$ are two type environments such that the fragment $\mathtt{S}$, named RT, compiles into $\mathtt{B}$ in $\Gamma_{\texttt{old}}$, that is, the judgment $\Gamma_{\texttt{old}} \vdash \mathtt{S} \rightsquigarrow \mathtt{B}$ can be derived. Then,*

$$\Gamma_{\texttt{new}} \vdash \mathtt{S} \rightsquigarrow \mathtt{B} \iff \begin{cases} \vdash \Gamma_{\texttt{new}} \diamond \\ \Gamma_{\texttt{new}}(\mathit{RT}) = \Gamma_{\texttt{old}}(\mathit{RT}) \\ \Gamma_{\texttt{new}} \vdash \mathit{Reqs}(\mathit{RT}, \Gamma_{\texttt{old}}) \end{cases}$$

Proof

The thesis follows by Theorem 4.2.4 and Theorem 4.2.8, using the same reasoning used to prove them.

□

By Theorem 4.2.9, recompiling an unchanged source `S` if and only if the new environment does not entail the requirements of `S` is both *sound* and *minimal*.

4.2.6 Incremental environment checking

Many judgments we have defined rely on a type environment Γ^{S} to be well-formed; that is, not containing cycles in the type hierarchy and respecting the rules on hiding/overriding. We now formalize this idea.

A typing environment Γ^{S} is well-formed, $\vdash \Gamma^{\mathrm{S}} \diamond$, if $\Gamma^{\mathrm{S}} \vdash \texttt{okOvr RT}$ can be derived for any of its types, Figure 4.22.

The judgment

$$\Gamma^{\mathrm{S}} \vdash \texttt{okOvr RT}$$

has the meaning "`RT` correctly extends its parent types (up to Object) in Γ^{S}". That is, the hierarchy of `RT` is acyclic and the Java rules on method overriding/hiding are respected. The metarules defining such a judgment are shown in Figure 4.22.

$$\frac{\Gamma^{\mathrm{S}} \vdash \mathtt{RT} : \langle \text{AFSS}{=}\_, \text{ANHS}{=}\_, \text{NOVR}{=}\_ \rangle}{\Gamma^{\mathrm{S}} \vdash \mathtt{okOvr}\ \mathtt{RT}}$$

$$\frac{\forall \mathtt{RT} \in \mathit{Def}(\Gamma^{\mathrm{S}})\ \Gamma^{\mathrm{S}} \vdash \mathtt{okOvr}\ \mathtt{RT}}{\vdash \Gamma^{\mathrm{S}} \diamond}$$

Figure 4.22: Well-formed standard environments

The idea behind our compilation strategy is to determine which (unchanged) sources have to be recompiled testing which assumptions, generated by a previous compilation, still hold in the new environment $\Gamma_{\mathtt{new}}$ (that is, the environment extracted by the updated fragments). Because the results of such tests are meaningful only if $\Gamma_{\mathtt{new}}$ is a well-formed environment, we need to check the well-formedness of $\Gamma_{\mathtt{new}}$ each time our compilation manager is run.

Checking the whole environment can be expensive; fortunately, if we know a previous well-formed environment $\Gamma_{\mathtt{old}}$, we can use it to avoid checking the "old" part of a new environment $\Gamma_{\mathtt{new}}$. That is, we can check only the "updated" part of $\Gamma_{\mathtt{new}}$, with respect to $\Gamma_{\mathtt{old}}$, instead of checking the whole $\Gamma_{\mathtt{new}}$.

Note that $\Gamma_{\mathtt{old}}$ can be *any* well-formed environment, although the "old" in $\Gamma_{\mathtt{old}}$ expresses the idea that an implemented compilation manager would probably save a type environment as soon as it is proved to be well-formed. In this way, the compilation manager can check any new environment $\Gamma_{\mathtt{new}}$ against $\Gamma_{\mathtt{old}}$ (instead of checking the whole $\Gamma_{\mathtt{new}}$) in the subsequent runs. When a new environment $\Gamma_{\mathtt{new}}$ is found to be well-formed, then it is saved to be used later as "$\Gamma_{\mathtt{old}}$", and so on.

Because any well-formed $\Gamma_{\mathtt{old}}$ can be used, requiring the knowledge of a "previous" well-formed type environment does not limit the applicability of our incremental checking: the first time our approach is used the empty environment, which is trivially well-formed, can be used as $\Gamma_{\mathtt{old}}$.

We now formalize these ideas.

Def. 4.2.10 *A type environment $\Gamma_{\mathtt{new}}$ is well-formed w.r.t. another type environment $\Gamma_{\mathtt{old}}$ iff the following conditions hold:*

[add] $\mathtt{RT} \in \mathtt{leaves}_{\Gamma_{\mathtt{new}}}(\mathit{Def}(\Gamma_{\mathtt{new}}) \setminus \mathit{Def}(\Gamma_{\mathtt{old}})) \implies \Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}$

[rmv] $\forall \mathtt{RT} \in \mathit{Def}(\Gamma_{\mathtt{old}}) \setminus \mathit{Def}(\Gamma_{\mathtt{new}}),$

$$\forall \mathtt{RT}' \in \mathit{Def}(\Gamma_{\mathtt{old}})\ \Gamma_{\mathtt{old}} \vdash \mathtt{RT}' \leq \mathtt{RT} \implies \begin{cases} \mathtt{RT}' \notin \mathit{Def}(\Gamma_{\mathtt{new}}) \text{ or} \\ \mathit{directSuper}(\Gamma_{\mathtt{new}}(\mathtt{RT}')) \in \mathit{Def}(\Gamma_{\mathtt{new}}) \end{cases}$$

[cng] $\mathtt{RT} \in \mathit{Def}(\Gamma_{\mathtt{old}}) \cap \mathit{Def}(\Gamma_{\mathtt{new}}),\ \Gamma_{\mathtt{old}}(\mathtt{RT}) \neq \Gamma_{\mathtt{new}}(\mathtt{RT}) \implies$
$\forall \mathtt{RT}' \in \mathtt{leaves}_{\Gamma_{\mathtt{new}}}(\mathtt{RT})\ \Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}'$

where:

$$\mathtt{leaves}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}) = \{\mathtt{RT}' | \Gamma^{\mathrm{S}} \vdash \mathtt{RT}' \leq \mathtt{RT}\ \wedge\ \forall \mathtt{RT}''\ \Gamma^{\mathrm{S}} \vdash \mathtt{RT}'' \leq \mathtt{RT}' \implies \mathtt{RT}'' = \mathtt{RT}'\}$$
$$\mathtt{leaves}_{\Gamma^{\mathrm{S}}}(\{\mathtt{RT}_1, \ldots, \mathtt{RT}_n\}) = \mathtt{leaves}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}_1) \cup \ldots \cup \mathtt{leaves}_{\Gamma^{\mathrm{S}}}(\mathtt{RT}_n)$$

$$\mathit{directSuper}([\text{TYPE}{=}\mathtt{class}, \text{AM}{=}\_, \text{CK}{=}\_, \text{PARENT}{=}\mathtt{C}', \text{IS}{=}\mathtt{I}_1 \ldots \mathtt{I}_n, \text{KSS}{=}\_, \text{FSS}{=}\_, \text{NHS}{=}\_]) = \{\mathtt{C}', \mathtt{I}_1, \ldots, \mathtt{I}_n\}$$
$$\mathit{directSuper}([\text{TYPE}{=}\mathtt{interface}, \text{AM}{=}\_, \text{IS}{=}\mathtt{I}_1 \ldots \mathtt{I}_n, \text{FSS}{=}\_, \text{NHS}{=}\_]) = \{\mathtt{I}_1, \ldots, \mathtt{I}_n\}$$

Lemma 4.2.11 *If RT is a supertype of RT′ in Γ^S, then $\Gamma^S \vdash$ `okOvr` RT′ implies $\Gamma^S \vdash$ `okOvr` RT.*

Proof

Trivial. □

Theorem 4.2.12 *If $\vdash \Gamma_{\mathtt{old}}\diamond$ holds, then $\Gamma_{\mathtt{new}}$ is well-formed w.r.t. $\Gamma_{\mathtt{old}} \Leftrightarrow\ \vdash \Gamma_{\mathtt{new}}\diamond$.*

Proof

⇒)

We must show that the judgment $\Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}$ is derivable for any $\mathtt{RT} \in \mathit{Def}(\Gamma_{\mathtt{new}})$. This can be proved by case analysis. Consider $\mathit{Def}(\Gamma_{\mathtt{new}})$ as the union of three disjoint sets: U, C and N. These sets contain, respectively, the unchanged, changed and new types in $\Gamma_{\mathtt{new}}$ with respect to $\Gamma_{\mathtt{old}}$. Formally:

$$U = \{\mathtt{RT} | \mathtt{RT} \in \mathit{Def}(\Gamma_{\mathtt{new}}) \cap \mathit{Def}(\Gamma_{\mathtt{old}}), \Gamma_{\mathtt{new}}(\mathtt{RT}) = \Gamma_{\mathtt{old}}(\mathtt{RT})\}$$
$$C = \{\mathtt{RT} | \mathtt{RT} \in \mathit{Def}(\Gamma_{\mathtt{new}}) \cap \mathit{Def}(\Gamma_{\mathtt{old}}), \Gamma_{\mathtt{new}}(\mathtt{RT}) \neq \Gamma_{\mathtt{old}}(\mathtt{RT})\}$$
$$N = \mathit{Def}(\Gamma_{\mathtt{new}}) \setminus \mathit{Def}(\Gamma_{\mathtt{old}})$$

Let us consider $N \cup C$ first. For any new type RT, contained in N, the judgment $\Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}$ is derivable by the hypothesis, see *[add]* of Definition 4.2.10, and Lemma 4.2.11. For any changed type RT, contained in C, the judgment $\Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}$ is derivable because of *[cng]* of Definition 4.2.10 and Lemma 4.2.11.

It remains to prove that the judgment is derivable for the unchanged types, contained in U. Since they are unchanged the *direct* supertypes of any type in U are the same in $\Gamma_{\mathtt{old}}$ and $\Gamma_{\mathtt{new}}$. Furthermore, each direct supertype of $\mathtt{RT} \in U$ must be contained in $\mathit{Def}(\Gamma_{\mathtt{old}})$ *and* in $\mathit{Def}(\Gamma_{\mathtt{new}})$. Indeed, if a direct supertype of RT, say RT′, were in $\Gamma_{\mathtt{old}}$ but not in $\Gamma_{\mathtt{new}}$, then by the hypothesis *[rmv]*:

- RT has been removed too, but this is impossible because $\mathtt{RT} \in U$, or
- $\mathit{directSuper}(\Gamma_{\mathtt{new}}(\mathtt{RT})) \in \mathit{Def}(\Gamma_{\mathtt{new}})$.

Hence, these direct supertypes must be contained in $\mathit{Def}(\Gamma_{\mathtt{old}}) \cap \mathit{Def}(\Gamma_{\mathtt{new}})$ which, by definition, is equal to: $U \cup C$. If a direct supertype is in U, then the same reasoning can be applied; so, for any $\mathtt{RT} \in U$, only two cases are possible:

- *all* supertypes of RT are in U; then, the hierarchy of RT has not changed and by the hypothesis $\vdash \Gamma_{\mathtt{old}}\diamond$ the judgment $\Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}$ is derivable too;
- there exists a supertype $\mathtt{RT}'$ of RT which is in C, whose subtypes till RT are in U. Then, by *[cng]* there exists a type $\mathtt{RT}''$ such that $\mathtt{RT}'' \leq \mathtt{RT}$ and $\Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}''$ holds. So, $\Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}$ is derivable too by Lemma 4.2.11.

$\Leftarrow$)

Requirements *[add]* and *[cng]*, of Definition 4.2.10, are trivially met because the judgment $\Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}$ is derivable for *any* $\mathtt{RT} \in \mathit{Def}(\Gamma_{\mathtt{new}})$ by the hypothesis $\vdash \Gamma_{\mathtt{new}}\diamond$.

Requirement *[rmv]* can be proved by contradiction. Suppose there exist $\mathtt{RT} \in \mathit{Def}(\Gamma_{\mathtt{old}}) \setminus \mathit{Def}(\Gamma_{\mathtt{new}})$, $\mathtt{RT}' \in \mathit{Def}(\Gamma_{\mathtt{old}})$ such that $\Gamma_{\mathtt{old}} \vdash \mathtt{RT}' \leq \mathtt{RT}$ and both:

- $\mathtt{RT}' \in \mathit{Def}(\Gamma_{\mathtt{new}})$
- $\mathit{directSuper}(\Gamma_{\mathtt{new}}(\mathtt{RT}')) \notin \mathit{Def}(\Gamma_{\mathtt{new}})$

This is absurd because, by hypothesis, the judgment $\Gamma_{\mathtt{new}} \vdash \mathtt{okOvr}\ \mathtt{RT}'$ can be derived and it could not be if $\mathit{directSuper}(\Gamma_{\mathtt{new}}(\mathtt{RT}'))$ would not be contained in $\mathit{Def}(\Gamma_{\mathtt{new}})$.

$\square$

4.3 Implementation issues

As we have proved for a substantial subset of Java, our recompilation strategy is both *sound* and *minimal*; from a theoretical point of view this is the best we can achieve. Yet, from a practical point of view, there is another point to ponder: the cost of checking whether the requirements Γ of a fragment S are entailed by a type environment $\Gamma^{\mathtt{S}}$. If this checking costed more than compiling the source S, then the whole idea would be useless.

A simple optimization is to use our recompilation strategy as a refinement of some other less precise (but faster) strategy; for instance, as a refinement for a cascading recompilation strategy à la *make* [Fel79]. In practice, though, what really matters is not to compile the minimum number of sources, but to have a fast *sound* recompilation strategy.

Because in the global cost of a recompilation we must take into account both the time spent in checking the entailment, for deciding which unchanged sources we have to recompile, and the time spent in recompiling the selected sources, choosing the minimum number of sources to be recompiled does not necessarily mean choosing the fastest recompilation strategy. Therefore, it makes sense to simplify the type assumptions, making them less "precise", in order to speed up the entailment checking step. Basically, trading a speed up in the average case with the possibility of performing some "useless" recompilations once in a while.

For instance, instead of using the abstract exception expressions to model all the possible cases in which different `throws` clauses do not affect reachability, we could check for equality of `throws` clauses and decide to recompile a fragment `S` when the clause of the most specific method for an invocation in `S` changes. Of course, this simplification needs to take into account the type hierarchy of the exceptions (because the meaning of a `throws` clause depends on that).

In the formal model we have not modeled `imports`, assuming all names to be fully-qualified. Of course, an actual implementation must keep track of the members of each package too, as a simple name may become ambiguous when a new type is introduced.

We now consider a series of issues which should help in simplifying, and so speeding up, the entailment checking step:

- Members of standard classes/interfaces are likely to remain the same, so one could ignore the type assumptions for method invocations/field accesses on standard classes (to be on the safe side, a global recompilation may be needed when the SDK is upgraded since new methods may affect overloading resolution and, less likely but not impossible, new fields may hide old ones).

- As said before, instead of evaluating exception expressions we could keep track of the `throws` clause of the most specific method for a method invocation and check for equality (as long as the exception hierarchy does not change).

- Ctc-fields are tricky: *if* they are correctly used, that is, as long as constants do not change, they are not a problem, and we do not need any type assumption for tracking ctc-fields. Evaluating their initialization expressions every time we extract a new environment seems, on the average, too expensive (after all, they are supposed not to change). We could cache their values and recalculate them only when necessary, but there are two simpler solutions which seem appropriate. We can either ignore them (assuming to be in a perfect world) or keep track of which fragment uses, directly or indirectly, a ctc-field and recompile every client each time a ctc-field changes.

The former solution is the easiest and fastest, but of course it is not sound. On the other hand, the latter solution is sound but may trigger unnecessary recompilations. A sensible tradeoff could be letting the user to decide (for instance via a compiler option) whether propagate the recompilation in these cases. The compiler manager may facilitate the user's choice issuing a warning each time a change in the definition of a ctc-field is detected.

A compiler manager for full Java, based on these ideas, is under development.

Chapter 5

Related work

5.1 Formalizations of Java

An important stream of research related to this thesis is devoted to the formal definition of Java (see [AF99] for a survey). As already mentioned, the type judgment which we consider at the source level is based on the many existing formal Java type systems, in particular those in [DE99]. For what concerns an integrated formal model covering all Java aspects, the most remarkable amount of work in this direction is that of Sophia Drossopoulou and her group. The already cited [DE99] provides a formal type system at the source level for a substantial subset $\mathrm{Java}^{\mathtt{s}}$ of Java and a translation of this language into a binary language $\mathrm{Java}^{\mathtt{b}}$, which is in turn a subset of a language $\mathrm{Java}^{\mathtt{r}}$ of run-time terms for which an operational semantics is given. This allows to prove type safety of the Java subset. In [DWE98] the focus is on binary compatibility. In [Dro01] a model is defined for dynamic loading and linking, distinguishing five components in a Java implementation: evaluation, resolution, loading, verification, and preparation, with their associated checks. These five together are proved to guarantee type soundness. This paper is the most important reference for the execution model of our small binary language given in Chapter 2; however, in our case the main aim is not to define a realistic model of the JVM, taking into account all features, but to show how absence of linkage errors can be guaranteed by a compilation schema, so we take a much more abstract view. Finally, [DVE00] enhances the previous formal description of Java in [DE99], introducing, among other improvements, an account of separate compilation. Indeed, type information used in typechecking $\mathrm{Java}^{\mathtt{s}}$ can also be extracted from the binary language $\mathrm{Java}^{\mathtt{r}}$, analogously to what we do in this paper by means of the $\mathcal{T}$ function. However, the judgment for typechecking source classes defined in [DVE00] do not correspond to separate compilation as happens in our framework, simply because its validity requires the type environment extracted from the compilation environment to be well-formed.

5.2 Separate compilation

The seminal paper on separate typecheck of fragments is [Car97]. There, the basic idea is to distinguish a phase of *intra-checking*, which models separate compilation, in which a single fragment is type-checked w.r.t. a typing environment (which expresses the interface of the fragment in terms of both imported and exported services), and a phase of *inter-checking* which models (static) linking, in which it is checked that all the fragments we want to link have been type-checked w.r.t. compatible type environments.

Formally[1], intra-checking is modeled by a judgment $\Gamma \vdash f : T$ (in [Car97] issues of code generation are avoided by always working at the source level, even when discussing linking), expressing that the fragment f has type T in the type environment Γ. Inter-checking takes place on *linksets* which are, roughly, collections of named fragment $x_i \mapsto \Gamma_i \vdash f_i : T_i^{i \in 1..n}$, and succeeds if and only if intra-checking succeeds (that is, each $\Gamma_i \vdash f_i : T_i$ holds) and, moreover, for each $j, k \in 1..n$, x_j has type T_j in Γ_k. This corresponds to require *exact* agreement among the actual interface of a fragment and that assumed in another: in realistic systems, this condition should be weakened, for instance requiring some subtyping relation.

As we discussed, Java has many features which make this view not immediately applicable: class files play the dual roles of interfaces (type environments) and object files; there is no separate linking phase, since linking takes place at run-time; compilers usually incorporate *some* inter-checks, but not enough to guarantee safe run-time linking.

In literature several interesting papers can be found on separate compilation for ML (see among many others [SA93, Ler94, Blu99]). All these papers clearly show that separate compilation in ML is not a simple issue, and for this reason, needs to be properly formalized. However, ML separate compilation is based on traditional static linking, therefore many problems arising in Java disappear in ML; for instance, the static type-checks proposed in [SA93] are sensible for a static linker, but cannot be performed at run-time by a virtual machine without seriously compromising efficiency. Furthermore, it seems that no unifying frameworks have been defined for investigating ML separate compilation, and in fact, this would be useful to compare all the technical results and to understand how they can contribute all together to the design of a better compiler/linker for ML. For instance, using the terminology used in our paper to model the overall compilation process, [SA93] is mainly concerned with the definition of the type extraction function, while [Ler94] with the typechecking of sources and [Blu99] with the definition of the dependency function.

[1] We use slightly different notations from [Car97].

5.3 Selective recompilation

The approach to selective compilation presented in Chapter 3 and Chapter 4, which extends the ideas in [AL03, Lag03, Lag04a], is similar to attribute recompilation, according to the classification given in [ATW94] - here attributes correspond to assumptions.

A very interesting framework for handling selective recompilation, in presence of these kind of inter-module dependencies and with the ability of trading off space usage, speed of processing and selectivity of invalidation is given in [CDG95]. In such a paper, the authors discuss how they developed a highly selective dependency mechanism for the Cecil object-oriented language [Cha93], and their method lookup filtering nodes play the same role of our method invocation assumptions.

In the context of procedural languages, [CKT86] discusses strategies to determine which procedures need to be recompiled after some changes, when inter-procedural optimizations introduce dependencies between sources.

The only Java-specific paper we know of is, as said, Dmitriev's [Dmi02], which describes a make technology, based on smart dependency checking, that aims to keep a project consistent while reducing the number of files to be recompiled.

Chapter 6

Conclusion and future work

The main contributions of this thesis can be summarized as follows.

- We have provided the first, at our knowledge, formalization of the Java typechecking and code generation process in the general case in which compilation takes place in a context of both source and binary fragments. This process was neither clearly specified in the Java official documentation, making the semantics of separate compilation implementation dependent, nor taken into account in previous type systems. More precisely, we have modeled the overall compilation process by the formal notion of compilation schema, in which the aspects which concern truly separate typechecking (source type judgment and binary type judgment) are isolated from the definition of dependencies, which models propagation of typechecking, and the extraction of the type information from fragments.

- By means of this model, we have been able to express, in a formal way, the fact that Java separate compilation is not type safe, in the sense that there is no guarantee that in running a successfully compiled class we get no linkage errors. This model has also permitted to show how type safety can be achieved by modifying two components of the SDK compilation schema. The ingredients are both stronger dependencies, which propagate typechecking to all classes which could be possibly loaded at run-time, and having a non trivial binary type judgment.

- We have defined an innovative type system where a single class declaration can be compiled in total isolation (true separate compilation), providing a set of type assumptions on missing classes. In Java separate compilation, instead, a class is typed in a global type environment containing full type information on used classes, extracted from available source and binary fragments. The relevance of this result is twofold. At theoretical level, it shows that the possibility to define type systems for Java-like languages which fit in the modular approach to typechecking based on intra-checking and inter-checking phases, as promoted by [Car97]. From the practical point of view, the type system can be fruitfully used to implement a selective recompilation strategy which is both sound and minimal, that is, a strategy equivalent to global recompilation which never triggers useless recompilations (i.e., recompilations which produce binaries equal to the existing ones). This strategy consists in generating, the first time a source `S` is successfully compiled into a binary `B`, the *weakest* type assumptions which describe the requirements for `S` to be compiled into that specific binary `B`. So, each time some sources are changed, we can decide whether an unchanged source has to be recompiled checking whether its assumptions still hold.

- Finally, we have extended the type system in order to handle a substantial subset of Java, obtaining a formal basis for a smart compilation manager for Java, which is currently under development. As side result, we have also obtained the most complete, at our knowledge, formal type system for Java covering many aspects which were not considered until now. The reason is probably the rather low-level nature of most of the previously uncovered features, which, however, become relevant for selective recompilation. Indeed, another important side result of our work is that we can formally express a criticism to the design of some Java features, e.g., compile-time constant fields and reachability of code; that is, that they badly interact with modularity. In our framework this is clearly illustrated by the involved type assumptions required to handle these two features.

There are two main directions for future work related with this thesis.

First, an interesting topic is the development of a formal framework for Microsoft's .NET analogous to that we have developed in Chapter 2 for Java separate compilation. Indeed, although .NET shares many features with Java, our model cannot properly model the novel concept of assembly, which allows to decouple the logical and physical notions of reusable types [Ric02]. Hence, our framework should be extended to take this extra level of indirection into account.

Second, a limitation of the type system for true separate compilation presented here is that, since in Java the same source fragment can generate different binary fragments depending on the context, it is not possible to infer the minimal set of assumptions needed for compiling a class by just inspecting the source code. This problem has been solved in this thesis by defining the minimal set of assumptions needed for generating a given binary code. While this approach, as illustrated above, works well for selective recompilation, where the minimal assumptions can be generated the first time a source is compiled, it prevents from having a type inference algorithm for Java-like languages. Hence, the direct use of this type system would put the burden of writing the type assumptions on programmers. This activity would be both tedious and rather time-consuming for real projects.

An interesting alternative direction is the development of a more expressive type system where we can infer infer the minimal set of assumptions needed for typechecking a class by just inspecting the source code, abstracting from the different binaries which would be generated in different contexts. This approach would require polymorphic types, hence it requires the introduction of a more abstract notion of bytecode, where type annotations are allowed to be polymorphic.

Bibliography

[AF99] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Number 1523 in Lecture Notes in Computer Science. Springer, 1999.

[AL03] D. Ancona and G. Lagorio. Stronger Typings for Separate Compilation of Java-like Languages. Technical report, DISI, March 2003.

[ALZ01] D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*, SIGPLAN Notices. ACM Press, October 2001.

[ALZ02] D. Ancona, G. Lagorio, and E. Zucca. A formal framework for Java separate compilation. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 609–635. Springer, 2002.

[ATW94] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[AZ04] D. Ancona and E. Zucca. Principal typings for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2004*, pages 306–317. ACM Press, January 2004.

[Blu99] M. Blume. Dependency analysis for standard ML. *ACM Transactions on Programming Languages and Systems*, 21(4):790–812, 1999.

[Car97] L. Cardelli. Program fragments, linking, and modularization. In *ACM Symp. on Principles of Programming Languages 1997*, pages 266–277. ACM Press, 1997.

[CDG95] Craig Chambers, Jeffrey Dean, and David Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *Proceedings: 17th International Conference on Software Engineering*, pages 221–230. IEEE Computer Society Press / ACM Press, 1995.

[Cha93] Craig Chambers. The cecil language: Specification and rationale. Technical report, University of Washington, March 1993.

[CKT86] Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM SIGPLAN Notices*, 21(7):58–67, July 1986.

[DE99] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer, 1999.

[Dmi02] M. Dmitriev. Language-specific make technology for the Java programming language. *ACM SIGPLAN Notices*, 37(11):373–385, 2002.

[Dro01] S. Drossopoulou. Towards an abstract model of Java dynamic linking and verfication. In R. Harper, editor, *TIC'00 - Third Workshop on Types in Compilation (Selected Papers)*, volume 2071 of *Lecture Notes in Computer Science*, pages 53–84. Springer, 2001.

[DVE00] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Dept. of Computing - Imperial College of Science, Technology and Medicine, September 2000.

[DWE98] S. Drossopoulou, D. Wragg, and S. Eisenbach. What *is* Java binary compatibility? In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, volume 33(10) of *SIGPLAN Notices*, pages 341–358, October 1998.

[Fel79] Stuart I. Feldman. Make-a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–65, 1979.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification, Second Edition*. Addison-Wesley, 2000.

[IPW99] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146, November 1999.

[Lag03] G. Lagorio. Towards a smart compilation manager for Java. In Blundo and Laneve, editors, *Italian Conf. on Theoretical Computer Science 2003*, number 2841 in Lecture Notes in Computer Science, pages 302–315. Springer, October 2003.

[Lag04a] G. Lagorio. Another step towards a smart compilation manager for Java. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, March 14-17, 2004*, pages 1275–1280. ACM, March 2004.

[Lag04b] G. Lagorio. Smart Recompilation for Java. Submitted for journal publication, March 2004.

[Ler94] X. Leroy. Manifest types, modules and separate compilation. In *ACM Symp. on Principles of Programming Languages 1994*, pages 109–122. ACM Press, 1994.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Second edition, 1999.

[Ric02] Jeffrey Richter. *Applied Microsoft .NET Framework*. Microsoft Press, 2002.

[SA93] Z. Shao and A.W. Appel. Smartest recompilation. In *ACM Symp. on Principles of Programming Languages 1993*, pages 439–450. ACM Press, 1993.

[SUN01] SUN Microsystems. The Java HotSpot Virtual Machine, 2001. Technical White Paper.

[Sym99] D. Syme. Proving Java type sound. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 83–118. Springer, 1999.

[vON99] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 119–156. Springer, 1999.