LJGS: Gradual Security Types for Object-Oriented Languages

Luminous Fennell
University of Freiburg, Germany
Email: fennell@informatik.uni-freiburg.de

Peter Thiemann
University of Freiburg, Germany
Email: thiemann@informatik.uni-freiburg.de

***Abstract*—LJGS is a lightweight Java core calculus with a gradual security type system. The calculus guarantees secure information flow for sequential, class-based, object-oriented programming with mutable objects and virtual method calls. An LJGS program is composed of fragments that are either checked statically against a security type system or that are checked dynamically by enforcing the no-sensitive-upgrade policy (NSU). Statically checked fragments have no run-time penalty whereas dynamically checked fragments rely on run-time security labels. The programmer marks the boundaries between static and dynamic checking with casts so that it is always clear whether a program fragment requires run-time checks. LJGS also provides a special branching statement to inspect the run-time label of a dynamic value, which makes its value available for static reasoning.**

LJGS requires security annotations on fields and methods. A field annotation either specifies a fixed static security level or it prescribes dynamic checking. A method annotation specifies a constrained polymorphic security signature. The types of local variables in method bodies are analyzed flow-sensitively and require no annotation.

We prove type soundness and non-interference for LJGS.

***Keywords*—gradual typing, security typing, Java, dynamic enforcement**

I. Introduction

Information-flow control (IFC) is a corner stone of language-based security. A typical IFC policy rules out the flow of information from classified sources to public sinks. The technical property aimed for is noninterference: any change in a classified source does not influence the public sinks. Noninterference comes in different flavors depending on the observational capabilities of an attacker (e.g., termination sensitive or not, batch or interactive).

There are also different kinds of IFC. A static system performs a static analysis, for instance using a security type system, and guarantees noninterference for analyzed programs. A dynamic system attaches run-time security labels to values, propagates them, and checks them at appropriate points during program execution. Dynamic systems can be more flexible, but they may have limitations in handling implicit flows, they impose a run-time overhead for manipulating the security labels, and they usually abort a program run on detecting a policy breach. Hybrid systems improve on both, in particular in their handling of implicit flows.

We propose a particular type-based amalgamation of static and dynamic IFC typing for Java, which is inspired by work on gradual typing [19], [11]. Gradual typing per se enables the composition of programs from typed and untyped fragments using suitable type casts at the interfaces. Gradual typing guarantees full compliance with the type system in the typed fragments, whereas untyped fragments may raise run-time type errors. Gradual security typing transposes this idea and its properties to an information flow setting.

We propose Lightweight Java with Gradual Security (LJGS) as a core calculus to substantiate our ideas. LJGS is a subset of sequential Java without exceptions and reflection inspired by Lightweight Java (LJ) [21]. Our specification is type-based in two respects: first, it takes the form of a type system; second, it assumes that the underlying program is well-typed according to some standard type system. Thus, LJGS is independent of specific Java typing features, in particular, it is compatible with generics. The LJGS types in this paper would translate to type annotations in a Java program.

Following Denning [7], LJGS specifies permissible information flows using a lattice $(\mathit{Sec}, \sqsubseteq)$ of *security levels* ranged over by A. For each field of a class, the programmer must specify a fixed security level or mark the field as dynamically checked. For each method, the programmer must supply a constrained polymorphic security method type signature (see examples in Section II). The signature relates the security types of the arguments, the result, and the effect on global variables with each other and with fixed levels that arise from field accesses. The security type of a local variable need not be declared; moreover, it can change during the method (it is *flow sensitive*).

Each value with a dynamic security type, in a dynamic field or in a temporarily dynamic local variable, carries a run-time security label that overapproximates its true security level. Updates to dynamic fields are checked using the no-sensitive-upgrade (NSU) policy [3], [27]; other sound dynamic upgrade policies can be used as well.

Technical results: Besides proving type preservation and progress (up to breaches of the security policy in dynamic updates), our key result is noninterference. To state this result precisely, we need to introduce our attacker model. An attacker has access to the public part of the heap, may construct the public arguments of any LJGS method, run it, and inspect the public part of the result and the heap afterwards. If the method diverges or aborts, the attacker receives no information. There is no way to obtain intermediate results. LJGS guarantees that such an attacker cannot learn information about non-public arguments or heap bindings. This guarantee corresponds

```
1  int max(int x, int y)
2    where { @x ≤ @return, @y ≤ @return } {
3    if (x ≤ y) {x = y;}
4    return x;
5  }

6  class Log { String buffer[LOW]; ... }
7  int maxWithMessage(Log log, int x, int y)
8    where { @x ≤ @return
9          , @y ≤ @return
10         , @log ≤ LOW
11   } and { LOW } {
12   if (x ≤ y) { x = y; }
13   log.buffer = "max was called";
14   return x;
15 }
```

```
16 class SecLog extends Log { String highBuf[HIGH]; }
17 int maxWithMessageDyn(SecLog log, int x, int y)
18   where { ⋆ ≤ @x
19         , ⋆ ≤ @y
20         , ⋆ ≤ @return
21         , log ≤ LOW }
22   and { ⋆ } {
23   if (x ≤ y) {
24     x = y;
25     iflabel (x ⊑ LOW; x_low) {
26       (⋆ ⇛ LOW) {
27         log.buffer = x_low + "is smaller";
28       }
29     }
30     (⋆ ⇛ HIGH) {
31       log.highBuf = ((HIGH ⇚ ⋆) x) + "is smaller";
32     }
33   }
34   return x;
35 }
```

Fig. 1. Examples of method definitions and signatures

to *batch termination insensitive non-interference* (BTINI), as defined by Askarov et al [2].

II. A TASTE OF LJGS

This section demonstrates the salient features of LJGS using illustrative code fragments in Figure 1. For these examples, we assume that the security lattice has two points, `LOW` and `HIGH`, where `HIGH` $\not\sqsubseteq$ `LOW`.

The two methods on the left illustrate how to specify information flow policies in LJGS via security signatures. The method `max` carries a polymorphic security signature in a **where** clause. It indicates that information may flow from the parameters `x` and `y` to the method's result by asserting that their security types are lower bounds for the type of the return value. In this signature, security types are purely symbolic: we write `@x` for the security type of the argument passed as parameter `x` and `@return` for the security type of the return value. As the symbolic types are not bounded by concrete security levels, `max` may be called in any context with arguments that satisfy the constraints. It may also be called with two dynamic arguments.

Method `maxWithMessage` comes with a signature that declares a global side effect to a `LOW` field in the **and** part of the **where** clause. The method performs the same computation as `max`. Additionally, it writes a message into the field `buffer` of a `Log` object. The definition of class `Log` declares that the field `buffer` has security level `LOW`. Consequently, calling `maxWithMessage` is only secure in contexts where the *program counter level* is `LOW`.[1] The type system of LJGS enforces this restriction by checking the declared global effect `{LOW}` against the *program counter type* at the call site. Additionally, the signature enforces that the `log` parameter is a low-security reference by upper-bounding it with `LOW`. Thus, `LOW` is an example of a fixed, static security type. The parameters `x` and `y` remain polymorphic, as they do not influence the execution of the global side effect.

While the effect of `maxWithMessage` is checked statically, the method `maxWithMessageDyn` illustrates the distinguishing feature of LJGS: the integration of dynamic and static enforcement of security policies. The idea of `maxWithMessageDyn` is to always write to a secret log, embodied by the field `highBuf` of security level `HIGH` in class `SecLog`, and to write to a public, low-security log, whenever that is possible. Such a combination, where publicly visible effects depend on the security level of an input, requires checking of security labels at run time. For example, in `maxWithMessage` either `x` or `y` could be confidential, resulting in a high-security program-counter type in the branches of the conditional **if** `(x ≤y) { ... }`. Thus, a security type checker would have to reject the low-security update in code like **if** `(x ≤y) { log.buffer = ... }`. Using gradual security enforcement, `maxWithMessageDyn` is able to write to low-security fields when called with low-security arguments and restrict output to high-security fields otherwise. To accommodate this feat, the signature of `maxWithMessageDyn` declares that parameters and result as well as the global effect are of *dynamic type* ($\star$). LJGS maintains run-time security labels for dynamically typed arguments and program counters and employs an non-sensitive upgrade (NSU) policy [3] to ensure secure information flow.

As dynamic arguments carry a run-time security label, we can compare the labels of `x` and `y` against the security level `LOW` in the **iflabel** statement in line 25. Only if `x` has a low security level, line 27 writes to the statically typed field `log.buffer`. The variable `x_low`, declared in the condition of the **iflabel** statement, is an unlabeled copy of `x` with type `LOW`, matching the type of `log.buffer`. The interaction of (labeled) dynamic entities and (unlabeled) static entities is mediated by casts. The *context cast* `(⋆ ⇛ LOW) { s }` (line 26) checks at run time that the current program counter label is `LOW` and raises a security violation exception, otherwise.[2] This cast may fail if the program counter at the call site is already `HIGH`. Similarly, the (safe) assignment to `log.highBuf` is guarded by a context-cast to `HIGH`. To store the value of x into the field `highBuf` of type `HIGH` (line 31), the *value cast* `(HIGH ⇚⋆) x` checks that the label of `x` is convertible to `HIGH`. Both casts do not fail if `HIGH` is the greatest element of the lattice.

Figure 2 shows how dynamic and polymorphic methods can be called. Using value casts, lines 37 and 38 create dynamic

[1] As usual, we refer to the confidentiality of information that the execution of a context depends on as the *program counter level*.

[2] In a cast, the $\Rightarrow$ symbol always specifies the direction of the conversion.

```
36  SecLog log = new SecLog();
37  int x = (⋆ ⇐ HIGH) secret;
38  int y = (⋆ ⇐ LOW) 42;
39  int r;
40
41  r = this.maxWithMessage(log, secret, 42);
42  r = this.maxWithMessage(log, x, y);
43
44  // type error: mix of static and dynamic arguments
45  // r = this.maxWithMessage(log, secret, y);
46
47  (LOW ⇛ ⋆) {
48    r = this.maxWithMessageDyn(log, x, y);
49  }
50  // type error: dynamic arguments required
51  // r = this.maxWithMessageDyn(log, secret, 42);
```

Fig. 2. Calling dynamic and polymorphic methods

TABLE I. SUMMARY OF THE BEHAVIOR OF MAXWITHMESSAGEDYN.

label of *pc*	label of x	effect
LOW	HIGH	update highBuffer
HIGH	HIGH	update highBuffer
LOW	LOW	update buffer, highBuffer
HIGH	LOW	error: illegal context cast from HIGH to LOW.

versions of the high-security value stored in variable `secret` and a low-security constant `42`, storing them in `x` and `y`, respectively. The *source type* of casts, here `HIGH` and `LOW`, respectively, indicates what run-time label to attach to the cast value. The polymorphic method `maxWithMessage` accepts arguments that are either all static or all dynamic (line 42 and 41). The results have a static type `HIGH` and dynamic type $\star$, respectively. Both calls will write to the low security field `log.buffer` and thus require a program counter of type `LOW`. Calling `maxWithMessage` with mixed static and dynamic arguments (line 45) fails with a type error; the result type depends on both inputs but at type-checking time as well as at run time the type of one of the inputs is unknown.

Calling `maxWithMessageDyn` with `x` and `y` in line 48 yields a dynamic result, as indicated by its signature. Furthermore, the call should happen in a dynamic context which is created with the context cast `(LOW ⇛⋆)`. However, calling `maxWithMessageDyn` with the original static values would fail with a type error as the arguments do not carry run-time security labels. Table I illustrates the method's effects when called with different dynamic labels for program counter and argument `x`. The first entry corresponds to the call in line 48 of Figure 2: the program counter (*pc*) is labelled `LOW` and the argument `x` is labelled `HIGH`. Nothing is written to `buffer` to avoid leaking the secret contained in `x`. A call with high-security *pc* and high-security argument has the same behavior. Calling the method with low-security *pc* and `x` safely writes to `buffer`, as intended. The last configuration calls the method with a high-security *pc* but a low-security argument. As the **iflabel** statement in `maxWithMessageDyn` only tests the label of `x` and not that of the program counter, the context cast in line 26 fails.[3]

The method calls and type errors of Figure 2 illustrate a strict separation of dynamic and static types. This principle allows for simple reasoning about the overhead of dynamic security checks. The fully static method call in line 41, for example, requires no dynamic checks. Even the calls to `maxWithMessageDyn` need not check updates to the fields of the statically typed object `log`. However, until now we have ignored the security type of the method call receiver, **this**. In object-oriented languages, virtual method calls in general create implicit flows from the method call receiver into the program counter and the return value of the called method. Always treating method call receivers as either static or dynamic is not satisfactory: if **this** has a static type, the call in line 42 is problematic as it yields a dynamic result. If **this** has dynamic type, then the static side effect of `maxWithMessage` interferes with a dynamic program counter. For these situations, LJGS supports a dedicated *public type* $\bullet$ that can act polymorphically as a low-security static type or the dynamic type. The run-time system ignores the security levels of values with public type. By assigning **this** the type $\bullet$, the method calls in Figure 2 type-check as intended. Thus, to be well-typed, the code in Figure 2 needs to placed in a method with a corresponding constraint for **this**:

```
void main() where {this ≤ •} and {LOW} {
  // ... <code of Figure 2>
}
```

The public type is intended for objects of classes like `Main`, whose prevalent purpose is not to store information but to provide methods. Usually, such objects are used rather as a context than as data. For simplicity, LJGS does not support creating objects of public type directly. Instead, variables of public type are introduced by casting low-security values. For example, assuming the methods defined in Figure 1 and `main` are part of the class `Main`, the following code correctly calls method `main` on a public object:

```
Main m = new Main[LOW]();
Main m_public = (• ⇐ LOW) m;
m_public.main();
```

It creates a `Main` instance of type `LOW` and applies a cast to $\bullet$.[4] Casts from `LOW` to $\bullet$ never fail, if `LOW` is the bottom element of the security lattice. An implementation could provide syntactic sugar for such public initializations.

III. SYNTAX OF LJGS PROGRAMS

The syntax of LJGS is inspired by Lightweight Java [21]: it supports imperative programming, structured control flow, inheritance, and (virtual) methods. LJGS extends Lightweight Java with annotations for security types and casts.

To better focus on the security aspects and to avoid notational clutter, we only write the annotations relevant for security typing, but we omit the Java types and signatures for fields, local variables, and methods.[5] Nevertheless, we assume that all programs are well-typed Java programs where types, signatures, and declarations of local variables are erased.

An LJGS program (see Figure 3) consists of a set of class definitions, *cld*, which in turn consist of field declarations,

[3] It is possible to add another branching statement to test for the label of a dynamic program counter. For simplicity, we omit it from the main discussion and present this extension in Appendix A.

[4] In constructor expressions, the security level of a new static instance is specified in square brackets after the class name.

[5] In an implementation, we would have to write the Java types and signatures in addition to the security annotations.

$$
\begin{array}{lll}
prog & ::= & cld_1 \,..\, cld_n \\
cld & ::= & \mathbf{class}\, C\, \mathbf{extends}\, cl\{F_1[a_1] \,..\, F_n[a_n]\, md_1 \,..\, md_m\} \\
cl & ::= & C \mid \mathbf{Object} \\
a, pc & ::= & \bullet \mid A \mid \star \\
md & ::= & M(var_1, \,..\, , var_n)\, \mathbf{where}\, Sig\, \mathbf{and}\, \mathcal{E}\{s\, \mathbf{return}\, y\} \\
x & ::= & var \mid \mathbf{this} \\
e & ::= & x \mid x.F \mid N \mid N_\star[A] \mid x + y \mid (a \Leftarrow a')x \\
s & ::= & var = e \mid x.F = y \mid var = x.M(y_1, \,..\, , y_n) \mid s; s \\
 & \mid & var = \mathbf{new}\, C[A](x_1, \,..\, , x_n) \\
 & \mid & var = \mathbf{new}_\star C[A](x_1, \,..\, , x_n) \\
 & \mid & \mathbf{if}\,(x == y)\{s\}\{s\} \mid \mathbf{while}\,(x == y)\{s\} \\
 & \mid & \mathbf{iflabel}\,(x \sqsubseteq A; var)\{s\}\{s\} \mid (a \Rrightarrow a)\{s\} \\
 & \mid & (a \Rrightarrow a)\{s\}
\end{array}
$$

Fig. 3. Syntax of LJGS

$F_i[a_i]$, and method declarations, md_i. Classes form a hierarchy with **Object** at its root. All method and field names are unique.

A field declaration relates a field name with a security type a. A security type is either the *public type* $\bullet$, a static security level A or the *dynamic type* $\star$. Security types are ranged over by a when referring to variable and field types and by pc (program counter) when referring to the current execution context.

A method definition declares the method name, M, a list of parameters, a security type signature Sig, an effect $\mathcal{E}$, a statement s that serves as method body, and a single local variable y for the return value. Method signatures are discussed in detail in Section IV. Further local variables are implicitly defined by their first assignment.

A variable x is either user-defined, var, or references the receiver of the method call, **this**. An expression e can be a variable access, x, field access $x.F$, a static or dynamic integer constant, N, or $N_\star[A]$, integer addition, $x + y$, or a value cast. A dynamic constant, $N_\star[A]$, carries an explicit security level that is used for run-time security enforcement. A cast expression $(a \Leftarrow a')x$ converts the value stored in variable x from *source type* a' to *destination type* a.

The first variant of a statement is a *local update*, $var = e$, that assigns the value of an expression e to a variable. A *field update* $x.F = y$ writes the value of y to field F of the object referenced by x. A *method call* $var = x.M(y_1, \,..\, , y_n)$ stores the result of calling method M with arguments $y_1, \ldots, y_n$ in variable var. Like the introduction of constants, construction of objects comes in two flavors, $var = \mathbf{new}\, C[A](x_1, \,..\, , x_n)$ and $var = \mathbf{new}_\star C[A](x_1, \,..\, , x_n)$, which respectively create a static or dynamic reference of security level A to a new object. Both construct an object of class C with fields initialized to the arguments $x_1, \ldots, x_n$. The **if** and **while** statements express structured control flow, as usual. The statement $\mathbf{iflabel}\,(x \sqsubseteq A; var)\{s_1\}\{s_2\}$ is a special conditional that branches on the dynamic security label of a value; execution continues with s_1 if x's security level is less than or equal to A and it continues with s_2, otherwise. The variable var declared in the condition has the static type A and holds the same value as x when s_1 is executed. Finally, the *context cast* statement $(pc \Rrightarrow pc')\{s\}$ embeds a computation with *inner* program counter type pc' into a context with *outer* program counter type pc.

$$
\begin{array}{lrcl}
\text{sig. constraints} & sc & ::= & sl \leq sl \mid sl \sim sl \mid sl \rightarrow_\star sl \\
\text{sig. level} & sl & ::= & a \mid @x \mid @\,\mathbf{return}
\end{array}
$$

Fig. 4. Components of method signatures

IV. Security Constraints and Typing Rules

In LJGS, a programmer specifies information-flow properties by providing suitable method signatures. As illustrated in Section II, a signature relates parameters, return values, and side-effects of methods with security types. The type system ensures that the signature specifications are adequate and that operations depending on statically typed parameters and fields have no security leaks. Its design also ensures that the security levels of statically typed values need not be tracked at run time.

Security types form a lower semi-lattice induced by the following ordering:

Definition 1 (Partial order on security types)**.**

$$
\frac{}{\bullet \leq a} \qquad \frac{A \sqsubseteq A'}{A \leq A'} \qquad \frac{}{\star \leq \star}
$$

By embedding the lattice of security levels into security types, LJGS supports the usual notion of security subtyping: values with a low security level are implicitly promoted to a higher one and contexts with a low-security program counter type admit computations that perform side effects on a higher security level. To ensure a clean boundary between static and dynamic code, implicit conversion between static types and the dynamic type is not allowed. However, the public security type $\bullet$ may act polymorphically as the dynamic type and the static type at the bottom of the security lattice ($\bot$).

A. Method Signatures

A method signature Sig is a set of *signature constraints* sc, defined in Figure 4. A single constraint relates its two *signature types*, sl_1 and sl_2, either by *subsumption*, $sl_1 \leq sl_2$, by *compatibility*, $sl_1 \sim sl_2$, or by *dynamic implication* $sl_1 \rightarrow_\star sl_2$. Signature types sl are literal security types a, or symbols for a method's formal parameters $@x$ (including the self-reference $@\mathbf{this}$), or its return value, $@\,\mathbf{return}$.

The constraint $sl_1 \leq sl_2$ ("sl_2 subsumes sl_1") specifies that sl_1 and sl_2 should be ordered according to Definition 1. For example, the constraint $\texttt{HIGH} \leq @\,\mathbf{return}$ requires the return value of a method to be a statically known security level that is at least HIGH. The constraint $sl_1 \sim sl_2$ ("sl_1 is compatible with sl_2") specifies there should be some relation between $sl_1 \sim sl_2$. For example, if sl_1 is dynamic then sl_2 should be dynamic or public. If both components are static, they need not be further related. Finally, the constraint $sl_1 \rightarrow_\star sl_2$ specifies that if sl_1 is dynamic then sl_2 should be dynamic or public. If sl_1 is not dynamic, then no restrictions are imposed on sl_2.

A method signature represents the information flow dependencies between parameters and return values: flows into the return value are represented by lower bounds on the return symbol, whereas flow restrictions on the arguments are represented by upper bounds on parameter symbols.

$$
\begin{array}{rrcl}
\text{typing constraints} & c & ::= & sec \leq sec \mid sec \sim sec \\
 & & \mid & sec \rightarrow_\star sec \\
\text{components} & sec & ::= & a \mid tvar \\
\text{type variables} & tvar & ::= & x \mid \mathbf{return} \mid \alpha \\
\text{pc type} & \gamma & ::= & \alpha \mid \bullet
\end{array}
$$

Fig. 5. Typing constraints

The effect annotation $\mathcal{E}$ on method definitions is a set of security types. If a security type is listed in $\mathcal{E}$, then the method may perform a side effect that leaks information into a (globally) accessible field of that type.

Compatibility and dynamic implication constraints rarely appear in signatures. As an example for the need of compatibility, consider a method, `const42`, that contains the same statements as `max`, but subsequently overwrites the result with a constant.

```
int const42(int x, int y)
  where { LOW ≤ @return , @x ∼ @y } {
  if (x ≤ y) {x = y;}
  x = 42;
  return x;
}
```

If `const42` is called with a dynamic first argument, the second argument needs to be dynamic or public to enable the run-time system to track implicit flows arising from the conditional. The signature expresses this requirement with the compatibility constraint $x \sim y$.

The following method, `returnY`, illustrates the need for dynamic implication constraints.

```
1 int returnY(bool h, int x, int y)
2   where {@h ≤ ⋆ , @y ≤ @return, h→⋆x } {
3   if (h) {
4     x = 5;
5   }
6   x = y;
7   return x;
8 }
```

If the boolean `h` is dynamic the update in line 4 requires an NSU check which compares the security label of the old value of `x` with that of `h`. This check is only possible when `x` also carries a dynamic label and thus the signature contains a dynamic compatibility constraint between `h` and `x`. As `x` is overwritten again in line 6 there are no further constraints involving `x`.

B. Typing Constraints and Constraint Interpretation

LJGS' typing rules, explained in detail in Section IV-C, generate sets of *typing constraints*, $\mathcal{C}$, and effects, $\mathcal{E}$, to represent the information flows in a method's body. Figure 5 defines the structure of typing constraints. Like signature constraints, typing constraints relate their *components*, sec, by subsumption, by compatibility, or by dynamic implication. Components are either security types or type variables. Type variables are parameters, x, the return variable, **return**, and anonymous type variables. In the typing rules, anonymous type variables represent the security types of local variables. We let α, β range over anonymous type variables. We write $\alpha^\dagger$ to denote a fresh anonymous type variable. Program counter types γ are either the public program counter type $\bullet$, or an anonymous type variable. Program counter types may also serve as the left-hand side component of a constraint.

A *signature instantiation*, $Sig[sl_1 \mapsto tvar_1, .., sl_n \mapsto tvar_n]$, interprets method signatures as typing constraints by replacing parameter symbols and return symbols with type variables. We write $\mathbf{constraints}\,(Sig)$ for the signature instantiation that maps all parameter symbols in Sig to corresponding parameter type variables and the return symbol to **return**.

We now formally define the interpretation of a constraint set.[6] In the following, let $\mathcal{C}$ range over sets of typing constraints.

Definition 2 (Assignment, solution, solvability)**.** An assignment is a total function from type variables to security levels. An assignment θ is extended to a total function from components to security levels, θ^*, by mapping security levels to themselves:

$$\theta^*(sec) = \begin{cases} a & \text{if } sec = a \\ \theta(tvar) & \text{if } sec = tvar \end{cases}$$

A particular assignment θ is a *solution* for a constraint set $\mathcal{C}$, written $\theta \models \mathcal{C}$, iff (i) for all constraints $sec_1 \leq sec_2 \in \mathcal{C}$, it holds that $\theta^*(sec_1) \leq \theta^*(sec_2)$, (ii) for all constraints $sec_1 \sim sec_2$, it holds that either $\theta^*(sec_1) \leq \theta^*(sec_2)$ or $\theta^*(sec_2) \leq \theta^*(sec_1)$, and (ii) for all constraints $sec_1 \rightarrow_\star sec_2$, it holds that if $\theta^*(sec_1) = \star$ then $\theta^*(sec_2) \in \{\star, \bullet\}$. In particular, constraints of the form $\bullet \leq sec$ impose no restrictions on solutions and compatibility constraints containing a static and a dynamic component (e.g., $\star \sim$ `LOW`) are never solvable.

Type checking of LJGS programs requires that the method signatures and effects subsume the typing constraints and effects generated for the corresponding method bodies. For signatures and typing constraints, subsumption amounts to checking entailment of constraints modulo anonymous type variables.

Definition 3 (Subsumption of signatures and typing constraints)**.** Let $\mathcal{C}_1$, $\mathcal{C}_2$ be two constraint sets. $\mathcal{C}_1$ *subsumes* $\mathcal{C}_2$, written $\mathcal{C}_2 <: \mathcal{C}_1$, if for each solution $\theta \models \mathcal{C}_1$ there exists an assignment θ' such that (i) $\theta' \models \mathcal{C}_2$, (ii) $\theta'(x) = \theta(x)$ for all parameters x, and (iii) $\theta'(\mathbf{return}) = \theta(\mathbf{return})$.

Let furthermore Sig_1, Sig_2 be two signatures. Sig_1 *subsumes* Sig_2, written $Sig_2 <: Sig_1$, if $\mathbf{constraints}\,(Sig_2) <: \mathbf{constraints}\,(Sig_1)$.

For example, the signature $\{\texttt{HIGH} \leq @\mathbf{return}\}$ which can be instantiated to $\{\texttt{HIGH} \leq \mathbf{return}\}$ subsumes the (more permissive) constraints $\{\texttt{LOW} \leq @\mathbf{return}\}$ and $\{\texttt{LOW} \leq \alpha, \alpha \leq \mathbf{return}, \texttt{HIGH} \leq \beta\}$.

For effects, subsumption amounts to checking subsumption contravariantly on the contained security types.

Definition 4 (Subsumption of effects)**.** Let $\mathcal{E}_1$, $\mathcal{E}_2$ be two effects. $\mathcal{E}_1$ *subsumes* $\mathcal{E}_2$, written $\mathcal{E}_2 <: \mathcal{E}_1$, if for all $a \in \mathcal{E}_2$ there exists $a' \in \mathcal{E}_1$ such that $a' \leq a$.

[6] Method signatures are interpreted by first instantiating them to typing constraints and then interpreting the result.

$\boxed{\Gamma, \alpha \vdash e : \mathcal{C}}$

RT-CONST-STATIC

$$\frac{}{\Gamma, \alpha \vdash N : \{\bot \sim \alpha\}}$$

RT-CONST-DYN

$$\frac{}{\Gamma, \alpha \vdash N_\star[A] : \{\star \sim \alpha\}}$$

RT-MOVE

$$\frac{}{\Gamma, \alpha \vdash x : \{\Gamma(x) \leq \alpha\}}$$

RT-GETFIELD

$$\frac{}{\Gamma, \alpha \vdash x.F : \{\Gamma(x) \leq \alpha, \mathbf{fsec}\,(F) \leq \alpha\}}$$

RT-PLUS

$$\frac{}{\Gamma, \alpha \vdash x + y : \{\Gamma(x) \leq \alpha, \Gamma(y) \leq \alpha\}}$$

RT-CAST

$$\frac{a \lesssim a'}{\Gamma, \alpha \vdash (a' \Leftarrow a)x : \{\Gamma(x) \leq a, a' \leq \alpha\}}$$

Fig. 6. Expression typing

$\boxed{a \lesssim a'}$

$$\frac{}{\star \lesssim a} \qquad \frac{}{a \lesssim \star} \qquad \frac{}{\bullet \lesssim A} \qquad \frac{}{\bot \lesssim \bullet}$$

Fig. 7. Castability

For example, the effect $\{\mathit{low}, \star\}$ subsumes $\{\mathit{high}, \star\}$ and $\{\mathit{low}, \mathit{high}, \star\}$. In contrast $\mathcal{E} = \{\mathit{low}\}$ does not subsume $\{\mathit{high}, \star\}$ as there is no type in $\mathcal{E}$ that is subsumed by $\star$.

C. Statement and Method Typing

The typing judgment for statements approximates the information flow in statements by generating constraints on type variables that represent the types of local variables or of the program counter. The statement typing judgment has the form $\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}$. It generates typing constraints $\mathcal{C}$ and effects $\mathcal{E}$. Statements are typed in a context with program counter type γ. Γ_1 and Γ_2 are typing environments. A statement is well-typed if it is possible to derive a typing for it that generates solvable typing constraints. The generated effect $\mathcal{E}$ has no impact on constraint solvability but is required to check a method's body against its signature (cf. Section IV-C2 below). The typing environments Γ_1 and Γ_2 are finite maps from local variables to type variables and thus allow to interpret the typing constraints with respect to local variables. Together with $\mathcal{C}$, the *initial environment* Γ_1 describes the typing constraints for local variables before s is executed, whereas the final environment Γ_2 describes local variable types after executing s.

1) Typing for Statements and Expressions: The typing judgment for expressions $\Gamma, \alpha \vdash e : \mathcal{C}$ determines constraints $\mathcal{C}$ for the expression's result. The type environment Γ provides the type variables for the variables occuring in e and the anonymous type variable α represents the type of the expression's result. We require that $\alpha \notin \mathrm{ran}(\Gamma)$. Figure 6 shows the rules for expression typing. Rules RT-CONST-STATIC and RT-CONST-DYN impose static and dynamic compatibility for static and dynamic constants, respectively. For variable access, rule RT-MOVE requires that the result type subsumes the type of the accessed local variable. Rule RT-GETFIELD requires that the result type subsumes the types of the accessed variable and field, where $\mathbf{fsec}\,(F)$ returns the declared type of field F. Rule RT-PLUS requires the result type to subsume the operand types. Rule RT-CAST requires that the result type subsumes the destination type of the cast and that the source type subsumes that of the accessed variable. Additionally the source type needs to be *castable* into the destination type, written $a \lesssim a'$. Castability rules out certain casts that are either unnecessary in the presence of security subtyping or are guaranteed to fail: Casts from A to A' are subtyping conversions when $A \sqsubseteq A'$ and impossible when $A \not\sqsubseteq A'$. Casts from A to $\bullet$ are also impossible when $A \neq \bot$. Figure 7 gives the corresponding rules for castability.

$\boxed{\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$

ST-LOCAL

$$\frac{\Gamma, \alpha^\dagger \vdash e : \mathcal{C} \qquad \Gamma_2 = \Gamma_1[\mathit{var} \mapsto \alpha^\dagger]}{\gamma \vdash \mathit{var} = e : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C} \cup \{\gamma \leq \alpha^\dagger, \gamma \rightarrow_\star \Gamma_1(\mathit{var})\}, \emptyset}$$

ST-PUTFIELD

$$\frac{\mathcal{C} = \{\Gamma(x) \leq \mathbf{fsec}\,(F), \Gamma(y) \leq \mathbf{fsec}\,(F), \gamma \leq \mathbf{fsec}\,(F)\}}{\gamma \vdash x.F = y : \Gamma \Rightarrow \Gamma, \mathcal{C}, \{\mathbf{fsec}\,(F)\}}$$

ST-NEW

$$\frac{\begin{array}{c} a_1 \,..\, a_n = \mathbf{fieldsecs}\,(C) \\ \mathcal{C} = \{\Gamma_1(x_1) \leq a_1, \,..\, , \Gamma_1(x_n) \leq a_n, A \leq \alpha^\dagger\} \\ \mathcal{C}' = \mathcal{C} \cup \{\gamma \leq \alpha^\dagger, \gamma \rightarrow_\star \Gamma_1(\mathit{var})\} \qquad \Gamma_2 = \Gamma_1[\mathit{var} \mapsto \alpha^\dagger] \end{array}}{\gamma \vdash \mathit{var} = \mathbf{new}\; C[A](x_1, \,..\, , x_n) : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \{A\}}$$

ST-NEW-DYN

$$\frac{\begin{array}{c} a_1 \,..\, a_n = \mathbf{fieldsecs}\,(C) \\ \mathcal{C} = \{\Gamma_1(x_1) \leq a_1, \,..\, , \Gamma_1(x_n) \leq a_n\} \\ \mathcal{C}' = \{\star \sim \alpha^\dagger, \gamma \leq \alpha^\dagger, \gamma \rightarrow_\star \Gamma_1(\mathit{var})\} \\ \Gamma_2 = \Gamma_1[\mathit{var} \mapsto \alpha^\dagger] \end{array}}{\gamma \vdash \mathit{var} = \mathbf{new}_\star\, C[A](x_1, \,..\, , x_n) : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \{\star\}}$$

Fig. 8. Statement typing: updates

Figures 8 and 9 define the statement typing rules. The rule for local assignment, ST-LOCAL, implements flow sensitivity by typing the expression with a fresh variable $\alpha^\dagger$ to obtain constraints $\mathcal{C}$. The final environment associates the destination variable var with $\alpha^\dagger$. Implicit flows are considered by requiring $\alpha^\dagger$ to subsume the program counter type. Adding a dynamic implication constraint between the program counter type and the old type variable of var ensures that NSU checks can be performed for dynamic program counters. Local updates are not visible outside of the method and thus generate no effects.

Rules ST-PUTFIELD types write operations to a field F. It requires that F's type subsumes the type of the source variable, the type of the accessed object reference x, as well as the type of the program counter. The statement's effect is also F's type. As no local variables are modified, initial and final environments are identical.

Rule ST-NEW types static allocation of objects. It requires

$\boxed{\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$

ST-CALL

$$\frac{\begin{array}{c} Sig = \mathbf{signature}\,(M) \qquad \mathcal{E} = \mathbf{effects}\,(M) \\ var_1, .., var_n = \mathbf{params}\,(M) \qquad \mathcal{C}' = Sig[@\mathbf{this} \mapsto \Gamma_1(x), @var_1 \mapsto \Gamma_1(x_1), .., @var_n \mapsto \Gamma_1(x_n), @\,\mathbf{return} \mapsto \alpha^\dagger] \\ \mathcal{C} = \mathcal{C}' \cup \{\Gamma_1(x) \leq \alpha^\dagger, \gamma \leq \alpha^\dagger, \gamma \rightarrow_\star \Gamma_1(var)\} \cup \bigsqcap_\gamma \mathcal{E} \qquad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger] \end{array}}{\gamma \vdash var = x.M(x_1, .., x_n) : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$$

ST-SEQ

$$\frac{\gamma \vdash s_1 : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}_1, \mathcal{E}_1 \qquad \gamma \vdash s_2 : \Gamma_2 \Rightarrow \Gamma_3, \mathcal{C}_2, \mathcal{E}_2}{\gamma \vdash s_1; s_2 : \Gamma_1 \Rightarrow \Gamma_3, \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{E}_1 \cup \mathcal{E}_2}$$

ST-IF

$$\frac{\begin{array}{c} \beta^\dagger \vdash s_1 : \Gamma_1 \Rightarrow \Gamma_2', \mathcal{C}_1, \mathcal{E}_1 \qquad \beta^\dagger \vdash s_2 : \Gamma_1 \Rightarrow \Gamma_2'', \mathcal{C}_2, \mathcal{E}_2 \\ \mathcal{C}' = \{\gamma \leq \beta^\dagger, \Gamma_1(x) \leq \beta^\dagger, \Gamma_1(y) \leq \beta^\dagger\} \\ \Gamma_2, \mathcal{C}'' = \Gamma_2' \sqcup \Gamma_2'' \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}' \cup \mathcal{C}'' \end{array}}{\gamma \vdash \mathbf{if}\,(x == y)\{s_1\}\{s_2\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}_1 \cup \mathcal{E}_2}$$

ST-WHILE

$$\frac{\begin{array}{c} \beta^\dagger \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \mathcal{E} \qquad \Gamma_3, \mathcal{C}'' = \Gamma_1 \sqcup \Gamma_2 \\ \mathcal{C} = \{\gamma \leq \beta^\dagger, \Gamma_1(x) \leq \beta^\dagger, \Gamma_1(y) \leq \beta^\dagger\} \cup \mathcal{C}' \end{array}}{\gamma \vdash \mathbf{while}\,(x == y)\{s\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$$

ST-IFLABEL

$$\frac{\begin{array}{c} \Gamma_1' = \Gamma_1[var \mapsto \alpha^\dagger] \\ \beta^\dagger \vdash s_1 : \Gamma_1' \Rightarrow \Gamma_2', \mathcal{C}_1, \mathcal{E}_1 \qquad \beta^\dagger \vdash s_2 : \Gamma_1 \Rightarrow \Gamma_2'', \mathcal{C}_2, \mathcal{E}_2 \\ a \in \{A, \star\} \qquad \mathcal{C}' = \{\gamma \leq \beta^\dagger, a \leq \beta^\dagger, \Gamma_1(x) \leq \star, A \leq \alpha^\dagger\} \\ \Gamma_2, \mathcal{C}'' = \Gamma_2' \sqcup \Gamma_2'' \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}' \cup \mathcal{C}'' \end{array}}{\gamma \vdash \mathbf{iflabel}\,(x \sqsubseteq A; var)\{s_1\}\{s_2\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}_1 \cup \mathcal{E}_2}$$

ST-CXCAST

$$\frac{\begin{array}{c} \beta^\dagger \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \mathcal{E} \\ \mathcal{C} = \{pc \leq \beta^\dagger, \gamma \leq pc'\} \cup \mathcal{C}' \qquad pc' \lesssim pc \end{array}}{\gamma \vdash (pc' \Rrightarrow pc)\{s\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \{pc'\}}$$

Fig. 9. Statement typing: branching, method calls and context casts

that the types declared for the fields of class C admit the arguments given to the constructor. The operation $\mathbf{fieldsecs}\,(C)$ looks up the declared field types. The result type is static and has to subsume the security level A mentioned in the constructor. The global effect is also $\{A\}$. Otherwise, the final environment and constraints are extended similarly as in rule ST-MOVE. Rule RT-NEW-DYN, for dynamic allocations, is analogous except that the result type and effect have to be dynamic.

Rule ST-CALL in Figure 9 covers method calls. It extracts the formal parameters, the signature, and the effect of the callee M with the operations $\mathbf{params}\,(M)$, $\mathbf{signature}\,(M)$, and $\mathbf{effects}\,(M)$. It instantiates the signature with the type variables corresponding to the arguments of the call. The return symbol is mapped to a fresh anonymous variable $\alpha^\dagger$. As in ST-PUTFIELD, the result type needs to subsume the program counter type and the type of the receiver object reference. Additionally, the program counter type needs to admit the effect $\mathcal{E}$ of the callee: Any security type in $\mathcal{E}$ should subsume the program counter type. The operation $\bigsqcap_\gamma \mathcal{E}$ generates the corresponding constraints:

$$\bigsqcap_\gamma \mathcal{E} := \{\gamma \leq a \mid a \in \mathcal{E}\}$$

The global effect is just $\mathcal{E}$. The final environment is updated with the result, as in ST-MOVE.

The remaining rules in Figure 9 are the inductive cases of statement typing. Rule ST-SEQ requires typings for the sequenced statements s_1 and s_2 under the same security context and combines the generated constraints and effects. The rule for conditionals, ST-IF, types the branches s_1 and s_2 under a fresh program counter variable $\beta^\dagger$. Types assigned to $\beta^\dagger$ need to subsume the old program counter type γ and the types of the condition variables x and y ($\mathcal{C}'$). The effects of the conditional is the union of the effects of both branches. The final environment is a join of the final environments of s_1 and s_2. The join operation $\Gamma_2' \sqcup \Gamma_2''$ generates additional constraints to be included in the final constraints $\mathcal{C}$.

Definition 5 (Join of typing environments)**.** An environment Γ together with a constraint set $\mathcal{C}$ form the join of two environments Γ_1 and Γ_2, written $\Gamma, \mathcal{C} = \Gamma_1 \sqcup \Gamma_2$, iff

$$\begin{aligned} \mathcal{C} = \quad & \{\Gamma_1(x) \leq \alpha_x \mid x \in \mathrm{dom}(\Gamma_1)\} \\ & \cup \{\Gamma_2(x) \leq \alpha_x \mid x \in \mathrm{dom}(\Gamma_2)\} \\ \Gamma = \{x \mapsto \alpha_x \mid x \in \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)\} & \end{aligned}$$

where $\{\alpha_x \mid x \in \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)\}$ is a set of fresh type variables, one for each variable in the domain of Γ_1 and Γ_2.

Rule ST-WHILE works similarly to rule ST-IF but joins the initial environment with the final environment of its body. Rule ST-IFLABEL requires that the tested variable x is public or dynamic and that the program counter type of the branches is either dynamic or a static type that subsumes A. The semantics of **iflabel** copy x to the variable var when x is less confidential than A. Otherwise var remains undefined. It is thus safe to give var the static type A by mapping it to the type variable $\alpha^\dagger$ and adding the constraint $A \leq \alpha^\dagger$. Otherwise rule ST-IFLABEL imposes similar restrictions to rule ST-IF. Rule ST-CXCAST deals with the cast between security contexts: The program counter type of the cast context needs to subsume the outer type pc' given in the cast. The outer type also defines the global effect of the statement. The body of the cast is typed under

$$
\begin{array}{lcl}
\varsigma, \ell, g & ::= & \mathbf{S} \mid \mathbf{D}(A) \mid \bullet \\
s & ::= & \ldots \mid \mathbf{done} \\
E & ::= & \langle\rangle \mid E; s \mid \mathbf{cx}\,[\ell]\{E\} \\
 & \mid & \mathbf{cx}\,[pc/\ell/g \Rrightarrow pc'/\ell/g']\{E\} \\
 & \mid & \mathbf{call}\,[M, var, x, y_1, .., y_n, L]\{E\} \\
rs & ::= & E(s) \\
v & ::= & rv[\varsigma] \\
rv & ::= & oref \mid N \\
oref & ::= & (op, PC) \\
L & ::= & \emptyset \mid L \oplus x \mapsto v \mid L \oplus x \mapsto \mathbf{null}
\end{array}
$$

Fig. 10. Dynamic domains and run-time statements

a fresh program counter type $\beta^\dagger$. The constraints require that $\beta^\dagger$ subsumes the inner type pc. Together with the typing rules ST-PUTFIELD and ST-CALL, this restriction guarantees that the global effect $\mathcal{E}$ of the body does not exceed $\{pc\}$. Also, pc' has to be castable to pc.

2) Method Typing: The method typing judgment $\vdash md$ asserts the adequacy of method signatures.

$$
\frac{\begin{array}{c}\Gamma_1 = [var_1 \mapsto var_1, .., var_n \mapsto var_n, \mathbf{this} \mapsto \mathbf{this}] \\ \bullet \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}' \qquad \alpha = \Gamma_2(y) \\ \mathcal{C} \cup \{\alpha \leq \mathbf{return}\} <: \mathbf{constraints}\,(Sig) \qquad \mathcal{E}' <: \mathcal{E}\end{array}}{\vdash M(var_1, .., var_n)\,\mathbf{where}\,Sig\,\mathbf{and}\,\mathcal{E}\{s\,\mathbf{return}\,y\}}
$$

The rule requires a typing of the method's body s. The program counter type is public to allow unconditional code in the method body to perform arbitrary local side effects. The initial typing environment maps the method's parameters to the corresponding parameter components. The typing of s yields constraints $\mathcal{C}$ and effect $\mathcal{E}$. The effect $\mathcal{E}$ should be subsumed by the declared effect $\mathcal{E}'$ while the generated constraints $\mathcal{C}$ should be subsumed by the method's signature.

As LJGS supports class-based subtyping, a method's signature has comply with the security policy of the method that it overrides. The following definition captures compliance.

Definition 6. The signature Sig and effects $\mathcal{E}$ of a method M *complies* to the class hierarchy,

1) if it does not override any other method, or
2) it overrides method M' with signature Sig' and effects $\mathcal{E}'$ and it holds that $Sig <: Sig'$ and $\mathcal{E} <: \mathcal{E}'$.

V. Dynamics

To enforce non-interference in dynamically checked code, LJGS' dynamics propagate and check run-time security labels. Before explaining the full operational semantics, we discuss the interpretation of security labels and their operations.

A. Security Labels

Security labels (see Figure 10) are the mirror image of security types: a security label is either the static label, $\mathbf{S}$, a dynamic label, $\mathbf{D}(A)$, carrying a security level or the *public label*, $\bullet$. The level of a static label is absent at run time because it has been checked by the type system already. The dynamic subset of security labels form a lattice based on the attached security levels. The entire set of labels forms a lower semi-lattice with $\bullet$ as bottom element, analogously to the semi-lattice of security types.

During the execution of an LJGS program, every value carries a *value label* ς. To check implicit flows, the semantics maintains two labels for each execution context, a *local program counter label*, ℓ, and a *global program counter label*, g. For value labels, this security level indicates the current run-time confidentiality of a labeled value. For program counter labels, it approximates the confidentiality of information that implicitly flows into the current context. Effectively, the dynamic label on a program counter gives a lower bound on the side effects.

When information flows into a value or context from multiple sources, the dynamic semantics employs a partial join operation on the labels involved. The judgment $\varsigma := \varsigma_1 \sqcup \varsigma_2$ determines the label ς that joins ς_1 and ς_2.

$$
\frac{}{\mathbf{S} := \mathbf{S} \sqcup \mathbf{S}} \qquad \frac{}{\mathbf{D}(A_1 \sqcup A_2) := \mathbf{D}(A_1) \sqcup \mathbf{D}(A_2)}
$$

$$
\frac{}{\varsigma := \bullet \sqcup \varsigma} \qquad \frac{}{\varsigma := \varsigma \sqcup \bullet}
$$

Joining static labels is trivial. Dynamic labels are joined by joining the security levels they contain. The public label is a neutral element for the join operation. Joining static and dynamic labels is undefined, as the security level of a static labeled entity cannot be recovered at run-time. A well typed LJGS program only performs well-defined joins.

B. Configurations and Reduction rules

The execution of a method manipulates *configurations* $rs/L/\mu$ of run-time statements rs, *stack frames* L, and *heaps* μ. Run-time statements, defined in Figure 10, consist of execution contexts E applied to source statements s. For technical reasons, the statements previously defined in Section III need to be extended with the marker statement $\mathbf{done}$. An execution context is a hole $\langle\rangle$, a sequence of an execution context and a source statement, a *run-time security context* $\mathbf{cx}\,[\ell]\{E\}$, a *cast context* $\mathbf{cx}\,[pc/\ell/g \Rrightarrow pc'/\ell'/g']\{E\}$, or a *calling context* $\mathbf{call}\,[M, var, x, x_1, .., x_n, L]\{E\}$. The run-time statement $\langle\mathbf{done}\rangle$ signals that execution of the current context is complete. A sequence context focuses the first component of a statement sequence. A run-time security context remembers a previous program counter label, for example during the execution of a conditional branch. A cast context stores the old and new program counter types and labels during the execution of a block subject to a context cast. A calling context stores the method name and stack frame of the callee and the return variable of the caller during a method call. Technicalities in the correctness proofs require that it also stores the method call receiver x and the call arguments $y_1, \ldots, y_n$.

A stack frame is a finite map from local variables to *values*. Stack frames have a fixed size and bind exactly the local variables that occur in the body of their method. Uninitialized variables are bound to $\mathbf{null}$.

A value consists of a *raw value* rv and value label ς. A raw value is either a *number* $N \in \mathbb{N}$ or an *object reference*, $oref$, which is composed of a pointer op and a *security region label* PC that identifies the security context of an object's allocation.

$\boxed{\ell; g \vdash rs/L/\mu \longrightarrow rs'/L'/\mu'}$

ESTEP-LOCAL
$$\frac{\ell \vdash e/L/\mu \Downarrow v/\mu' \qquad \ell \vdash var/L \Downarrow \mathbf{ok} \qquad L' = L[var \mapsto v]}{\ell; g \vdash \langle var = e\rangle/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L'/\mu'}$$

ESTEP-NEW
$$\frac{v_1 = L[x_1] \quad .. \quad v_n = L[x_n] \qquad oref' = \mathit{fresh}(A, \mu) \qquad v' = oref'[\mathbf{S}] \qquad \mu' = \mu \oplus oref' \mapsto \{C, v_1 .. v_n\} \qquad \ell \vdash var/L \Downarrow \mathbf{ok} \qquad L' = L[var \mapsto v']}{\ell; g \vdash \langle var = \mathbf{new}\ C[A](x_1, .., x_n)\rangle/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L'/\mu'}$$

$\boxed{\ell \vdash e/L/\mu \Downarrow v/\mu'}$

STEP-UPD-PLUS
$$\frac{N[\varsigma_1] = L[x] \qquad N'[\varsigma_2] = L[y] \qquad \varsigma' := \varsigma_1 \sqcup \varsigma_2 \qquad \varsigma := \varsigma' \sqcup \ell}{\ell \vdash x + y/L/\mu \Downarrow N + N'[\varsigma]/\mu}$$

STEP-UPD-CAST
$$\frac{rv[\varsigma] = L[x] \qquad a' \Rrightarrow a \vdash \varsigma \Rrightarrow \varsigma' \qquad \varsigma'' := \varsigma' \sqcup \ell}{\ell \vdash (a \Leftarrow a')x/L/\mu \Downarrow rv[\varsigma'']/\mu}$$

Fig. 11. Dynamics: local updates (excerpt)

A heap is a finite map from *references* to *objects*. An object consists of its run-time class C and the values of its fields.

The judgment $\ell; g \vdash rs/L/\mu \longrightarrow rs'/L'/\mu'$ defines the small step reduction of a configuration under the local program counter label ℓ and global program counter label g. Figures 11, 12, 13, and 14 show the most interesting reduction rules. Appendix B contains the full definition of the semantics.

Figure 11 shows the rules for local updates and static allocations. Local updates are factored in two stages: first the judgment $\ell \vdash e/L/\mu \Downarrow v/\mu'$ evaluates expression e to a result and an updated heap. Rule STEP-UPD-PLUS illustrates evaluation. It reads the operands from the stack frame L, and returns a raw value, here $N + N'$, that has an updated security label attached. The updated label is the join of the operands' labels and the program counter label. Then, the judgment $\ell \vdash var/L \Downarrow \mathbf{ok}$ performs an NSU check for an update of variable var in stack frame L under local program counter label ℓ. Other expressions work similarly. A static allocation additionally extends the heap with a fresh reference. It sets the allocation level of the fresh reference to the security level A mentioned in the constructor. In well-typed programs, A always subsumes the program counter type of the context. Allocation of dynamic objects (not shown) works similarly. A cast expression converts the value label of its subject according to the label conversion judgment $a \Rrightarrow a' \vdash \varsigma \Rrightarrow \varsigma'$, defined in Figure 13. Label conversion changes a label ς that corresponds to a cast's source type a to a label ς' corresponding to the destination type a'. A trivial cast results in a trivial label conversion. The conversion from a static type to the dynamic type creates a dynamic label that contains the source type as security level. A static-to-dynamic conversion always succeeds in a well-typed LJGS program. A conversion from dynamic to a static type A' returns the static label, if the dynamic source label contains a security level subsumed by A. Otherwise, the static-to-dynamic conversion fails. The public type may be converted to a dynamic type and a low-security label or any static type. Converting to the public type requires either a static source type of bottom or a dynamic bottom label.

Figure 12 defines the NSU check for local variables that rule ESTEP-LOCAL requires. Under static or polymorphic local

$\boxed{\ell \vdash var/L \Downarrow \mathbf{ok}}$
$$\frac{\ell \in \{\bullet, \mathbf{S}\}}{\ell \vdash var/L \Downarrow \mathbf{ok}} \qquad \frac{L[var] = \mathbf{null}}{\mathbf{D}(PC) \vdash var/L \Downarrow \mathbf{ok}}$$
$$\frac{L[var] = rv'[\mathbf{D}(A)] \qquad PC \sqsubseteq A}{\mathbf{D}(PC) \vdash var/L \Downarrow \mathbf{ok}}$$

Fig. 12. Dynamics: run-time checks

$\boxed{a' \Rrightarrow a \vdash \varsigma' \Rrightarrow \varsigma}$
$$\frac{}{a \Rrightarrow a \vdash \varsigma \Rrightarrow \varsigma} \qquad \frac{}{A \Rrightarrow \star \vdash \varsigma \Rrightarrow \mathbf{D}(A)}$$
$$\frac{A \sqsubseteq A'}{\star \Rrightarrow A' \vdash \mathbf{D}(A) \Rrightarrow \mathbf{S}} \qquad \frac{}{\bullet \Rrightarrow A \vdash \varsigma \Rrightarrow \mathbf{S}}$$
$$\frac{}{\bullet \Rrightarrow \star \vdash \varsigma \Rrightarrow \mathbf{D}(\bot)} \qquad \frac{}{\bot \Rrightarrow \bullet \vdash \mathbf{S} \Rrightarrow \bullet}$$
$$\frac{}{\star \Rrightarrow \bullet \vdash \mathbf{D}(\bot) \Rrightarrow \bullet}$$

Fig. 13. Label conversion

program counter labels, or when var is uninitialized, the check always succeeds. Otherwise the previous value stored in var has to carry a dynamic label that is at least as secure as the program counter label. The rule for global updates (not shown) works similarly to rule ESTEP-LOCAL. It employs an NSU check for the heap that uses the global program counter label instead of the local one.

Figure 14 shows illustrative cases of reductions that create modified contexts. Rule ESTEP-IF-TRUE covers if-statements where the branch condition is satisfied. Selecting a branch potentially creates an implicit flow from the condition to the execution context of the branch. Thus, the reduction results in a run-time security context that stores a program counter label

$\boxed{\ell; g \vdash rs/L/\mu \longrightarrow rs'/L'/\mu'}$

ESTEP-IF-TRUE
$$\frac{rv[\varsigma_1] = L[x] \qquad rv[\varsigma_2] = L[y] \qquad \varsigma' := \varsigma_1 \sqcup \varsigma_2 \qquad \ell'' := \ell \sqcup \varsigma'}{\ell; g \vdash \langle \mathbf{if}\,(x == y)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\ell'']\{\langle s_1\rangle\}/L/\mu}$$

ESTEP-IFLABEL-DYN-TRUE
$$\frac{rv[\mathbf{D}(A')] = L[x] \qquad A' \sqsubseteq A \qquad L[var] = \mathbf{null} \qquad L' = L[var \mapsto rv[\mathbf{S}]]}{\mathbf{D}(A''); g \vdash \langle \mathbf{iflabel}\,(x \sqsubseteq A; var)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\mathbf{D}(A'' \sqcup A)]\{\langle s_1\rangle\}/L/\mu}$$

ESTEP-CXCAST
$$\frac{pc \Rrightarrow pc' \vdash \ell \Rrightarrow \ell' \quad pc \Rrightarrow pc' \vdash g \Rrightarrow g'}{\ell; g \vdash \langle (pc \Rrightarrow pc')\{s\}\rangle/L/\mu \longrightarrow \mathbf{cx}[pc/\ell/g \Rrightarrow pc'/\ell'/g']\{\langle s\rangle\}/L/\mu}$$

ESTEP-CX-STEP
$$\frac{g' := g \sqcup \ell' \qquad \ell'; g' \vdash E(s)/L/\mu \longrightarrow E'(s')/L'/\mu'}{\ell; g \vdash \mathbf{cx}\,[\ell']\{E(s)\}/L/\mu \longrightarrow \mathbf{cx}\,[\ell']\{E'(s')\}/L'/\mu'}$$

ESTEP-CALL
$$\frac{\begin{array}{c} v_1 = L[x_1] \quad .. \quad v_n = L[x_n] \qquad oref[\varsigma] = L[x] \\ \ell' := \ell \sqcup \varsigma \qquad \{C, v_1 .. v_n\} = \mu[oref] \qquad M'(var_1, .., var_m)\,\mathbf{where}\,Sig\,\mathbf{and}\,\mathcal{E}\{s\,\mathbf{return}\,y\} = \mathbf{dispatch}\,(C, M) \\ L' = \mathbf{initframe}\,(M, \mathbf{this} \mapsto v, var_1 \mapsto v'_1, .., var_m \mapsto v'_m) \end{array}}{\ell; g \vdash \langle var = x.M(x_1, .., x_n)\rangle/L/\mu \longrightarrow \mathbf{cx}\,[\ell']\{\,\mathbf{call}\,[M', var, x, x_1, .., x_n, L']\{\langle s\rangle\}\}/L/\mu}$$

ESTEP-CALLING-STEP
$$\frac{\bullet; g \vdash E(s)/L'/\mu \longrightarrow E'(s')/L''/\mu'}{\ell; g \vdash \mathbf{call}\,[M', var, x, x_1, .., x_n, L']\{E(s)\}/L/\mu \longrightarrow \mathbf{call}\,[M', var, x, x_1, .., x_n, L'']\{E'(s')\}/L/\mu'}$$

ESTEP-CALLING-DONE
$$\frac{\begin{array}{c} M'(var_1, .., var_m)\,\mathbf{where}\,Sig\,\mathbf{and}\,\mathcal{E}\{s\,\mathbf{return}\,y\} = \mathbf{definition}\,(M) \\ rv[\varsigma] = L'[y] \qquad \varsigma' := \varsigma \sqcup \ell \qquad L' = L[var \mapsto rv[\varsigma']] \qquad \ell \vdash var/L \Downarrow \mathbf{ok} \end{array}}{\ell; g \vdash \mathbf{call}\,[M, var, x, var'_1, .., var'_n, L']\{\langle \mathbf{done}\rangle\}/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L'/\mu}$$

Fig. 14. Dynamics: entering modified contexts (excerpt)

ℓ'' which includes the value labels of the operands and the label of the outer context. The rules for failing conditions and while loops (not shown) are similar. The **iflabel** statement checks the dynamic label of its operand value and copies that value to an uninitialized, static variable that may be used in the "true" branch of the statement. Rule ESTEP-IFLABEL-DYN-TRUE covers the case where the label satisfies the condition in dynamic contexts. The resulting run-time security context is joined with the security level mentioned in the condition. The reductions for **iflabel** statements for failing conditions and in static contexts (not shown) are similar. Context casts, covered by rule ESTEP-CXCAST, determine the local and global program counter labels for executing their body with the same label conversion judgment used for value casts. A cast execution context stores the converted and original types and labels. Reduction under a run-time security context, covered by rule ESTEP-CX-STEP, uses the stored, modified program counter label. The global program counter label is joined with the stored label such that global effects also respect the augmented context. When a run-time security context is completely executed, it is discarded (not shown). Cast contexts (not shown) work similarly, but use the stored converted labels to reduce their bodies. Sequence execution contexts, also not shown, completely execute the first context before focusing the second, as expected.

Rule ESTEP-CALL, the reduction rule for method calls, sets up the context to evaluate the body of callee M. It retrieves the method arguments from the stack frame and determines the run-time class C from the callee's object. As M is potentially overloaded, it looks up the concrete implementation method M' via the operation $\mathbf{dispatch}\,(C, M)$. The operation $\mathbf{initframe}(M, \ldots)$ initializes the stack frame for M by mapping the parameter variables of M to the argument values of the method call and **this** to the callee's object. The reduction results in a calling context wrapped in a security context. The security context captures the implicit flow from the object to the program counter of the called method. The calling context stores the new stackframe and contains the method body in its hole. Reduction steps under calling contexts, covered by rule ESTEP-CALLING-STEP, take place under the global program counter label of the caller and under a polymorphic local program counter label. The public label corresponds to the public program counter type in the method typing rule described in Section IV-C. It permits to call method with only local implicit flows, as method `max` of Figure 1, to pass the NSU check on updates regardless of the global program counter label. When the method call is completed, rule ESTEP-CALLING-DONE copies the result value from the callee's frame to the caller's return variable. As the return variable is updated, the rule also performs an NSU check.

VI. Correctness

To guarantee that the statics and dynamics presented in the previous sections enforce security according to our attacker model, we need to prove termination insensitive non-interference. As the dynamics only check the security labels of

dynamically typed values, non-interference also relies on the soundness of the type system, which guarantees that statically typed data is not responsible for program crashes caused by security violations. We always implicitly assume that an LJGS program is well-formed according to the standard typing principles of Java. In particular we rely on the fact that no variables or object fields are accessed uninitialized. Also, we implicitly assume that all method signatures comply to the class hierarchy in an LJGS program, according to Definition 6. Proof sketches of the theorems stated in this section can be found in Appendix C.

A. Soundness of the type system

Simple typing judgments for values and annotations connect static and dynamic domains.

Definition 7 (Typing of Dynamic Domains)**.** A security label ς has type a, written $\vdash \varsigma : a$, if either (i) $\varsigma = \bullet$, (ii) $\varsigma = \mathbf{D}(A)$ and $a = \star$, or (iii) $\varsigma = \mathbf{S}$ and $a = A$.

A heap μ is well-typed, written $\vdash \mu$, if for all field values it contains, the value label has the type of the corresponding field declaration.

A stack frame L is well-typed under constraints $\mathcal{C}$ and environment Γ, written $\Gamma, \mathcal{C} \vdash L$ if for all bindings $x \mapsto rv[\varsigma]$ in L there exists an a such that $\mathrm{SAT}(\mathcal{C} \cup \{\Gamma(x) \leq a\})$ and $\vdash \varsigma : a$. This judgment essentially checks that dynamically typed variables have either dynamic or public labels.

Security constraints $\mathcal{C}$ type the program counter label ℓ with a program counter type γ, written $\mathcal{C} \vdash \gamma : \ell$, if either (i) $\mathrm{SAT}(\mathcal{C})$, $\ell = \bullet$, and $\gamma = \bullet$, (ii) $\ell = \mathbf{D}(PC)$ and $\mathrm{SAT}(\mathcal{C} \cup \{\gamma \leq \star\})$, or (iii) $\ell = \mathbf{S}$ and $\mathrm{SAT}(\mathcal{C} \cup \{\gamma \leq A\})$.

The type soundness and non-interference lemmas require a typing judgment for execution contexts, $\gamma, \ell, g \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'$. The judgment describes the typing environment of the execution context, and, given its program counter labels ℓ, g, it defines the program counter labels ℓ', g' that are active at its hole. The constraints, effect and environments that make up a context typing are the same as for statement typing. Figure 15 shows the context typing rules for holes, security contexts, and contexts casts. The other rules can be found in Appendix B. The rule for holes, ET-HOLE, is trivial. Rule ET-CX ensures that the body of a security evaluation context does not lower the program counter security level by generating a corresponding constraint for program counter types and by checking subsumption of dynamic program counter labels. The rule also ensures that the global program counter label always subsumes the local one. Rule ET-CXCAST captures the relation between program counter labels, program counter types, and effects of the body and context after a valid context cast. They correspond exactly to the requirements of typing rule ST-CXT-CAST and reduction rule ESTEP-CXCAST.

The typing of configurations relies on the typing of execution contexts:

$$\boxed{\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'}$$

$$\frac{\Gamma_1, \mathcal{C} \vdash L \qquad \vdash \mu \qquad \gamma, \ell, g \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g' \qquad \gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} \qquad \mathcal{C} \vdash \gamma : \ell \qquad \mathcal{C} \vdash \gamma : g \qquad \ell \leq g}{\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'}$$

It combines the requirements for well typed stack-frames, heaps, execution contexts, and statements with the constraint that the global program counter label needs to subsume the local one. The satisfiability of the generated constraints is implied by well-typed stack frames.

As usual for a gradually typed system, a well typed LJGS program guarantees a refined progress property. While (security related) run-time errors in statically typed code are ruled out, a program may run into a *dynamically stuck* configuration where dynamically typed code goes wrong.

Definition 8 (Dynamically stuck)**.** A well-typed configuration $\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'$ is *dynamically stuck* iff it attempts either:

1) a local sensitive upgrade, that is $s = (var = e)$, $\ell' = \mathbf{D}(PC)$, $L[var] = rv[\varsigma]$, $\varsigma \in \{\bullet, \mathbf{D}(A')\}$ and $\mathbf{D}(PC) \not\leq \varsigma$,
2) a global sensitive upgrade, that is $s = (x.F = y)$, $g' = \mathbf{D}(PC)$, $L[x] = oref[\varsigma]$, $\varsigma \in \{\bullet, \mathbf{D}(A'')\}$, $\mathbf{getfield}\,(F, \mu[oref]) = rv[\varsigma']$,$\varsigma' \in \{\bullet, \mathbf{D}(A')\}$ and $\mathbf{D}(PC) \not\leq \varsigma'$ or $\varsigma \not\leq \varsigma'$
3) an insecure value cast from dynamic to static, $s = (var = (a \Leftarrow \star)x)$, $a \in \{\bullet, A\}$, $L[x] = rv[\mathbf{D}(A')]$, and $a \not\leq A$, or
4) an insecure context cast from dynamic to static $s = ((\star \Rrightarrow a)\{s'\})$, $\ell' = \mathbf{D}(PC)$, $a \in \{\bullet, PC\}$and $PC \not\sqsubseteq A$

The progress result for LJGS states that well typed programs that are not able to make a reduction step are either $\mathbf{done}$, or dynamically stuck.

Theorem 1 (Progress)**.** *For any well typed configuration $\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'$ one of the following three cases holds: (i)$E(s) = \langle \mathbf{done} \rangle$, (ii) the configuration is dynamically stuck, or (iii) there exists $E'(s'), L', \mu'$ such that $\ell; g \vdash E(s)/L/\mu \longrightarrow E'(s')/L'/\mu'$*

Configuration typing allows us to state and prove a straightforward preservation theorem:

Theorem 2 (Preservation)**.** *Given a well typed configuration $\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'$ and a reduction step $\ell; g \vdash E(s)/L/\mu \longrightarrow E'(s')/L'/\mu'$ then there exists Γ_1', $\mathcal{C}'$, ℓ'', g'', and $\mathcal{E}'$ such that $\gamma, \ell, g \vdash E'(s')/L'/\mu' : \Gamma_1' \Rightarrow \Gamma_2, \mathcal{C}', \mathcal{E}' | \ell'', g''$, $\mathcal{C}' \subseteq \mathcal{C}$, and $\mathcal{E}' \subseteq \mathcal{E}$.*

B. Non-Interference

The non-interference theorem for LJGS states that methods run under *low-equivalent environments*, that is, environments that an attacker cannot distinguish, produce *low-equivalent results*. First we precisely define the notion of *low-equivalence*. In the following, we assume that *low* is the upper bound of security levels that an attacker can observe. We refer to a security level A as *high* if $A \not\sqsubseteq$ *low*.

Definition 9 (Low-equivalent objects, heaps and stack-frames)**.** Two objects Obj_1, Obj_2 are low-equivalent if they only differ in the values stored in high-security fields.

Two heaps μ_1, μ_2 are low-equivalent if (i) their domains agree on the references (op, PC) with allocation level $PC \sqsubseteq$

ET-HOLE

$$\frac{}{\gamma, \ell, g \vdash \langle\rangle : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell, g}$$

ET-CX

$$\frac{g \leq g' \qquad \ell' \leq g' \qquad \ell \leq \ell' \qquad \beta^\dagger, \ell', g' \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \mathcal{E}|\ell'', g'' \qquad \mathcal{C} = \mathcal{C}' \cup \{\gamma \leq \beta^\dagger\}}{\gamma, \ell, g \vdash \mathbf{cx}\,[\ell']\{E\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell'', g''}$$

ET-CXCAST

$$\frac{pc \Rrightarrow pc' \vdash \ell \Rrightarrow \ell' \qquad pc \Rrightarrow pc' \vdash g \Rrightarrow g' \qquad pc \lesssim pc' \qquad \beta^\dagger, \ell', g' \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \mathcal{E}'|\ell'', g'' \qquad \mathcal{C}'' = \{\gamma \leq pc, pc' \leq \beta^\dagger\}}{\gamma, \ell, g \vdash \mathbf{cx}\,[pc/\ell/g \Rrightarrow pc'/\ell'/g']\{E\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}' \cup \mathcal{C}'', \{pc\}|\ell'', g''}$$

Fig. 15. Context typing rules (excerpt)

low, and (ii) the objects stored at those references are low-equivalent.

Two environments L_1, L_2 are low-equivalent with respect to environment Γ, and solution θ, written $L_1 \equiv_{\Gamma,\theta,low} L_2$ if (i) $\mathbf{dom}\,(L_1) = \mathbf{dom}\,(L_2)$, (ii) $L_1[\mathbf{this}] = L_2[\mathbf{this}]$, and (iii) for all $var \in \mathbf{dom}(L_1)$ either

1) $L_1[var] = L_2[var]$,
2) $\theta(\Gamma(var)) = A$, and $A \not\sqsubseteq low$
3) $\theta(\Gamma(var)) = \star$ and $L_1[var] = rv[\mathbf{D}(A_1)]$, $L_2[var] = rv[\mathbf{D}(A_2)]$, and $A_1 \not\sqsubseteq low$ and $A_2 \not\sqsubseteq low$

With these definitions, the non-interference theorem for LJGS is as follows:

Theorem 3 (Non-interference)**.** *Let $E(s)/L_1/\mu_1$, $E(s)/L_2/\mu_2$ be two configurations with typings $\gamma, \ell, g \vdash E(s)/L_i/\mu_i : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$, ($i \in \{1, 2\}$) and solution $\theta \models \mathcal{C}$ where $L_1 \equiv_{\Gamma_1,\theta,low} L_2$ and $\mu_1 \equiv_{low} \mu_2$. Given two executions $\ell; g \vdash E(s)/L_i/\mu_i \longrightarrow^* \langle\mathbf{done}\rangle/L_i'/\mu_i'$ then $L_1' \equiv_{\Gamma_2,\theta,low} L_2'$ and $\mu_1' \equiv_{low} \mu_2'$.*

VII. Related Work

There is a large body of prior work on security type systems that ultimately goes back to Denning and Denning's classic paper on information flow security [8]. We focus on discussing works on security type systems and dynamic security enforcement designed for "main-stream" programming languages and defer the reader to the overview articles of Sabelfeld and Myers [17] and Hedin and Sabelfeld [13] for other aspects of language based security.

FlowCaML [15] is an ML dialect supporting static polymorphic security types with security constraints similar to those of LJGS. FlowCaML additionally supports higher-order types and complete type inference. Sun, Banerjee and Naumann describe a modular polymorphic type system for a Java-like, object oriented language [22]. The polymorphic method signatures are comparable to the static fragment of LJGS. Their system additionally supports class definitions with security type parameters and full type inference. Both approaches seems compatible with LJGS and we plan to investigate how they could be adapted for gradual security typing in future work. Barthe et al describe an information flow type system for Java Bytecode and a corresponding develop a corresponding certified type checker [6]. Their system supports Objects, virtual, monomorphic methods, exceptions and arrays. JIF [14] is an extension to Java with static security types and first-class dynamic security-labels. In JIF, security types may depend on dynamic labels and the interaction of labels and types is verified statically during type checking. In contrast, LJGS does not restrict dynamically labeled values statically but enforces non-interference at run-time. Another language that support statically checked, first-class security labels in the style of JIF is Grabowski and Beringer's DSD [12].

LJGS' run-time security enforcement for values with dynamic labels is an adaption of Austin and Flanagan's technique for purely dynamic information flow control technique with no-sensitive-upgrade policy [3]. Extensions of this approach, like permissive upgrade or faceted execution [4], [5], as well as approaches for hybrid information flow control based on dynamic labels [16] are compatible with LJGS.

The original work on gradual typing [25], [20] focuses on simple types with extensions like refinement predicates, polymorphism [1], and union types [24]. More recently, researchers started to gradualize type systems that check properties unrelated to the structure of values, like type annotations [11], ownership [18], typestate [26], and session types [23]. Gradual security type systems also fall into this category. Disney and Flanagan study gradual security types for a pure lambda calculus [9] and Fennell and Thiemann describe a system for a calculus with ML style references [10]. Compared with our treatment of object fields, Fennell and Thiemann's calculus treats mutable references in a more liberal way because it admits casts between reference types with static and dynamic content. However, this liberality comes at the cost of requiring pervasive run-time labelling even for static values and it blurs the separation of static and dynamic code, which runs contrary the execution model and design goal of LJGS.

VIII. Conclusion and Future Work

LJGS is a sequential Java core calculus with gradual security typing and branching on dynamic labels. The calculus strictly separates statically verified and dynamically checked code which enables running statically checked code without run-time security labels. Methods have polymorphic security signatures and can accept static or dynamic arguments in a suitable execution context.

There are several avenues for future work. We plan to implement type checking and run-time enforcement for (sequential) Java based on the principles of LJGS. With this implementation we want to investigate the practicality of the type system for realistic applications and to evaluate different compilation- and execution strategies for dynamic code. We

also want to extend the system with security type parameters for classes and methods, type inference, and hybrid monitoring of dynamically typed code.

REFERENCES

[1] A. Ahmed, R. B. Findler, J. Matthews, and P. Wadler. Blame for all. In *Proceedings of the 1st Workshop on Script to Program Evolution*, pages 1–13, Genova, Italy, 2009. ACM.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In S. Chong and D. A. Naumann, editors, *PLAS*, pages 113–124, Dublin, Ireland, June 2009. ACM.

[4] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 3:1–3:12, New York, NY, USA, 2010. ACM.

[5] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In J. Field and M. Hicks, editors, *Proc. 39th ACM Symp. POPL*, pages 165–178, Philadelphia, USA, Jan. 2012. ACM Press.

[6] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. *Mathematical Structures in Computer Science*, 23(5):1032–1081, 2013.

[7] D. Denning. A lattice model of secure information flow. *Comm. ACM*, 19(5):236–242, 1976.

[8] D. Denning and P. Denning. Certification of programs for secure information flow. *Comm. ACM*, 20(7):504–513, 1977.

[9] T. Disney and C. Flanagan. Gradual information flow typing. In *STOP*, 2011.

[10] L. Fennell and P. Thiemann. Gradual security typing with references. In V. Cortier and A. Datta, editors, *CSF*, pages 224–239, New Orleans, LA, USA, 2013. IEEE.

[11] L. Fennell and P. Thiemann. Gradual typing for annotated type systems. In Z. Shao, editor, *ESOP'14*, Lecture Notes in Computer Science, Grenoble, France, Apr. 2014. Springer.

[12] R. Grabowski and L. Beringer. Noninterference with dynamic security domains and policies. In A. Datta, editor, *Advances in Computer Science - ASIAN 2009. Information Security and Privacy, 13th Asian Computing Science Conference, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5913 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2009.

[13] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *2011 Marktoberdorf Summer School*. IOS Press, 2011.

[14] A. C. Myers. JFlow: Practical mostly-static information flow control. In A. Aiken, editor, *Proc. 26th ACM Symp. POPL*, pages 228–241, San Antonio, Texas, USA, Jan. 1999. ACM Press.

[15] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1):117–158, Jan. 2003.

[16] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, pages 186–199. IEEE Computer Society, 2010.

[17] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[18] I. Sergey and D. Clarke. Gradual ownership types. In *21th European Symposium on Programming (ESOP 2012)*, Tallinn, Estonia, Apr. 2012. Springer.

[19] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 2–27, Berlin, Germany, July 2007. Springer.

[20] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Sept. 2006.

[21] R. Strnisa and M. J. Parkinson. Lightweight java. *Archive of Formal Proofs*, 2011, 2011.

[22] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In R. Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2004.

[23] P. Thiemann. Gradual typing for session types. In E. Tuosto and M. Maffeis, editors, *TGC*, volume ? of *LNCS*, Rome, Italy, 2014. Springer. to appear.

[24] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In P. Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 395–406, San Francisco, CA, USA, Jan. 2008. ACM Press.

[25] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. 18th ESOP*, volume 5502 of *LNCS*, pages 1–16, York, UK, Mar. 2009. Springer.

[26] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In *ECOOP*, volume 6813 of *LNCS*, pages 459–483, Lancaster, UK, 2011. Springer.

[27] S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell, Ithaca, NY, USA, 2002.

APPENDIX A
BRANCHING ON PROGRAM COUNTER LABELS

The **ifpc** statement allows branching on program counter labels.

$$s ::= \ldots \mid \mathbf{ifpc}\,(A)\{s_1\}\{s_2\}$$

If the current dynamic program counter label is less confidential than A, then s_1 is executed under a static program counter type A. Otherwise s_2 is executed under a dynamic program counter type. Adding this branching statement allows to rewrite method `maxWithMessageDyn` of Figure 1 such that it never fails with a security exception:

```
1  int maxWithMessageDyn(SecLog log, int x, int y)
2    where { ⋆ ≤ @x
3          , ⋆ ≤ @y
4          , ⋆ ≤ @return
5          , log ≤ LOW }
6    and { ⋆ } {
7    if (x ≤ y) {
8      x = y;
9      ifpc(LOW) {
10       iflabel (x ⊑ LOW; x_low) {
11          log.buffer = x_low + "is smaller";
12     }
13     (⋆ ⇛ HIGH) {
14          log.highBuf = ((HIGH ⇚ ⋆) x) + "is smaller";
15     }
16   }
17   return x;
18 }
```

The update to `buffer` in line 11 is now only executed if the local and global program counter labels are (smaller than) `LOW`. Thus, in calls described by the last line of Table I, instead of an error, no write to `buffer` would occur. Also, no context cast is required for the update to `buffer`.

The typing rule for **ifpc** is as follows:

ST-IFPC

$$\frac{\begin{array}{c}\beta^\dagger \vdash s_1 : \Gamma_1 \Rightarrow \Gamma_2', \mathcal{C}_1, \mathcal{E}_1 \\ \gamma \vdash s_2 : \Gamma_1 \Rightarrow \Gamma_2'', \mathcal{C}_2, \mathcal{E}_2 \qquad \mathcal{C}' = \{\gamma \leq \star, A \leq \beta^\dagger\} \\ \Gamma_2, \mathcal{C}'' = \Gamma_2' \sqcup \Gamma_2'' \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}' \cup \mathcal{C}''\end{array}}{\gamma \vdash \mathbf{ifpc}\,(A)\{s_1\}\{s_2\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}_1 \cup \mathcal{E}_2}$$

It requires a program counter type of dynamic or static and types the "true" branch under a static program counter of level A.

Figure 16 gives the reduction rules for **ifpc**. In case the context label is smaller than A the statement reduced to a suitable cast context that contains s_1. Otherwise it is simply reduced to s_2.

APPENDIX B
COMPLETE STATIC AND DYNAMIC SEMANTICS

Figures 17, 18, 19, 20, and 21 give the complete dynamic semantics of LJGS. Figures 22, 23, 24, and 25 give the complete typing rules of LJGS.

APPENDIX C
CORRECTNESS

A. Progress

The proof is by induction on the typings of execution context E. The interesting causes for a configuration to be stuck are

1) an undefined join operation on labels, and
2) a failing NSU check
3) a failing cast conversion

In all cases the constraints generated by well-typed configuration ensure that all join operations required by the semantics are defined. This fact is essentially captured by the following lemma:

Lemma 1. *If $\vdash \varsigma : \theta(\alpha)$, $\vdash \varsigma' : \theta(\alpha')$, $\theta \models \mathcal{C}$ and $\{\alpha \leq \beta, \alpha' \leq \beta\} \subseteq \mathcal{C}$ then there exists ς'' such that $\varsigma'' := \varsigma \sqcup \varsigma'$. If $\vdash \varsigma : \theta(\alpha)$, $\vdash \ell : \theta(\gamma)$, $\theta \models \mathcal{C}$ and $\{\gamma \leq \alpha\} \subseteq \mathcal{C}$ then there exists ς'' such that $\varsigma' := \varsigma \sqcup \ell$.*

The lemma is easily verified by checking the definition of solvability.

By checking the definitions it is evident that the NSU check only fails for dynamic updates that are covered by *dynamically stuck*. Inspecting the definitions of *castability* and *label conversion*, it is clear that all failing well-typed casts fall under *dynamically stuck*.

B. Preservation

The proof is by induction on the reduction step. Most cases are straightforward. In cases that reduce to security contexts, like ESTEP-IF-TRUE, a weakening lemma is required, if the statement that is embedded in the context (the "true-branch" in case ESTE-IF-TRUE) has a high-security program counter type. The lemma asserts contravariant subtyping for program counter types.

Lemma 2 (Context weakening)**.** *If $\beta^\dagger \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}$ then $\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C} \cup \{\gamma \leq \beta^\dagger\}, \mathcal{E}$.*

C. Non interference

The proof is by induction on execution context. Most inductive cases are straightforward using the induction assumption and the type preservation result.

The interesting base-cases are those for branching statements and method calls. They require two confinement lemmas to show secure execution for the body of the context. Confinement states that the low-security parts of a configuration do not change when executing code with high-security global or local contexts.

Definition 10 (High security global context)**.** Given a well-typed configuration $\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$, g and $\mathcal{E}$ form a *high-security global context* if either (i) $g = \mathbf{S}$ and for all $PC \in \mathcal{E}$ it holds that $PC \not\sqsubseteq$ *low* or (ii) $g = \mathbf{D}(PC)$ and $PC \not\sqsubseteq$ *low*.

Lemma 3 (Global confinement)**.** *Given a well-typed configuration $\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$ and a reduction step $\ell; g \vdash E(s)/L/\mu \longrightarrow E'(s')/L'/\mu'$ then $\mu \equiv_{low} \mu'$ if g and $\mathcal{E}$ form a high-security global context.*

Definition 11 (High security local context)**.** Given a well-typed configuration $\beta, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$ with solution $\theta \models \mathcal{C} \cup \bigsqcap_\beta \mathcal{E}$, ℓ, and θ form a *high-security local context* if either (i) $\ell = \mathbf{S}$, $PC = \theta(\beta)$ and $PC \not\sqsubseteq$ *low* or (ii) $\ell = \mathbf{D}(PC)$ and $PC \not\sqsubseteq$ *low*.

ESTEP-IFPC-TRUE

$$\frac{\ell \leq \mathbf{D}(A)}{\ell; g \vdash \langle \mathbf{ifpc}\,(A)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\star/\ell/g \Rrightarrow A/\mathbf{S}/\mathbf{S}]\{\langle s_1\rangle\}/L/\mu}$$

ESTEP-IFPC-FALSE

$$\frac{\ell \not\leq \mathbf{D}(A)}{\ell; g \vdash \langle \mathbf{ifpc}\,(A)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \langle s_2\rangle/L/\mu}$$

Fig. 16. Dynamics for ifpc

Lemma 4 (Confinement)**.** *Given a well-typed configuration* $\gamma, \ell, g \vdash E(s)/L/\mu : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}|\ell', g'$ *with solution* $\theta \models \mathcal{C} \cup \prod_\gamma \mathcal{E}$ *where* ℓ *and* θ *for a high-security local context and a reduction step* $\ell; g \vdash E(s)/L/\mu \longrightarrow E'(s')/L'/\mu'$ *then* $L \equiv_{\Gamma_2,\theta,low} L'$ *and* $\mu \equiv_{low} \mu'$

The confinement lemmas are proven by induction on the reduction. Most cases are straightforward, given the definition for solvability of constraints. Also, as confinement is a generalization of global confinement, lemma 4 relies on lemma 3.

Proving non-interference for cases of branching statements like $\langle \mathbf{if}\,(x == y)\{s_1\}\{s_2\}\rangle$ requires to distinguish whether execution enter a high-security context or not, that is whether the conditions depends on high-security values or not. In case it does not, the low-equivalence assumption for stack frames guarantees that executions chooses the same branch for both configurations. The result then follows from the induction assumption. In case the condition depends on high-security values, the executions may choose different branches. We can still follow that $\langle \mathbf{if}\,(x == y)\{s_1\}\{s_2\}\rangle/L_i/\mu_i$ reduces to $\mathbf{cx}[\ell_i]\{\langle s_j\rangle\}/L_i/\mu_i$, where $j \in \{1, 2\}$ and $\langle s_j\rangle/L_i/\mu_i$ is in a local (and global) high-security context. As the semantics is deterministic, we can follow that $\langle s_j\rangle/L_i/\mu_i$ reduces to $\langle \mathbf{done}\rangle/L_i'/\mu_i'$. The result then essentially follows from lemma 4.

For the case of method calls, $\langle var = x.M(x_1, .., x_n)\rangle/L_i/\mu_i$, we need to distinguish whether $L_1[x] = L_2[x]$. If so, the configurations reduce to $\mathbf{cx}\,[\ell]\{\,\mathbf{call}\,[M, var, x, x_1, .., x_n, L_i']\{\langle s\rangle\}\}/L_i/\mu_i$. As the semantics is deterministic we know that there is an execution from $\langle s\rangle/L_i'/\mu_i$ to $\langle \mathbf{done}\rangle/L_i''/\mu_i'$. By induction assumption, L_1'' and L_2'', as well as μ_1' and μ_2' are low-equivalent, from which we can follow the result. If $L_1[x] \neq L_2[x]$, then we can follow from the low-equivalence assumption that x contains high-security information. Thus the configurations reduce to $\mathbf{cx}\,[\ell]\{\,\mathbf{call}\,[M, var, x, x_1, .., x_n, L_i']\{\langle s_i\rangle\}\}/L_i/\mu_i$ where $\langle s_i\rangle/L_i'/\mu_i$ is in a high-security local and global context. Again we have that $\langle s_i\rangle/L_i'/\mu_i$ reduces to $\langle \mathbf{done}\rangle/L_i''/\mu_i'$. By lemma 3, μ_1' and μ_2' are low-equivalent. When the method returns with ESTEP-CALLING-DONE, we have $L_i'' = L_i[var \mapsto v_i]$. As the method call executes in a locally high context, the update of return variable var does not impede low-equivalence, and thus L_1'' and L_2'' are low-equivalent.

$\boxed{\ell; g \vdash rs/L/\mu \longrightarrow rs'/L'/\mu'}$

ESTEP-LOCAL
$$\frac{\ell \vdash e/L/\mu \Downarrow v/\mu' \qquad \ell \vdash var/L \Downarrow \mathbf{ok} \qquad L' = L[var \mapsto v]}{\ell; g \vdash \langle var = e\rangle/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L'/\mu'}$$

ESTEP-NEW
$$\frac{\begin{array}{c} v_1 = L[x_1] \quad .. \quad v_n = L[x_n] \\ oref' = fresh(A, \mu) \qquad v' = oref'[\mathbf{S}] \qquad \mu' = \mu \oplus oref' \mapsto \{C, v_1 .. v_n\} \qquad \ell \vdash var/L \Downarrow \mathbf{ok} \qquad L' = L[var \mapsto v'] \end{array}}{\ell; g \vdash \langle var = \mathbf{new}\, C[A](x_1, .., x_n)\rangle/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L'/\mu'}$$

ESTEP-NEW-DYN
$$\frac{\begin{array}{c} v_1 = L[x_1] \quad .. \quad v_n = L[x_n] \qquad \mathbf{D}(A') := \mathbf{D}(A) \sqcup \ell \\ oref' = fresh(A', \mu) \qquad v' = oref'[\mathbf{D}(A')] \qquad \mu' = \mu \oplus oref' \mapsto \{C, v_1 .. v_n\} \qquad \ell \vdash var/L \Downarrow \mathbf{ok} \qquad L' = L[var \mapsto v'] \end{array}}{\ell; g \vdash \langle var = \mathbf{new}_\star\, C[A](x_1, .., x_n)\rangle/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L'/\mu'}$$

ESTEP-PUTFIELD
$$\frac{\begin{array}{c} rv'[\varsigma_1] = L[x] \qquad oref'[\varsigma_2] = L[y] \\ Obj = \mu[oref'] \qquad \varsigma' := \varsigma_1 \sqcup \varsigma_2 \qquad \varsigma := \varsigma' \sqcup \ell \qquad \mu' = \mu[oref' \mapsto Obj[F \mapsto rv'[\varsigma]]] \qquad g \vdash oref[\varsigma]/F/\mu \Downarrow \mathbf{ok} \end{array}}{\ell; g \vdash \langle x.F = y\rangle/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L/\mu'}$$

ESTEP-SEQ
$$\frac{}{\ell; g \vdash \langle s_1; s_2\rangle/L/\mu \longrightarrow \langle s_1\rangle; s_2/L/\mu}$$

ESTEP-CALL
$$\frac{\begin{array}{c} v_1 = L[x_1] \quad .. \quad v_n = L[x_n] \qquad oref[\varsigma] = L[x] \\ \ell' := \ell \sqcup \varsigma \qquad \{C, v_1 .. v_n\} = \mu[oref] \qquad M'(var_1, .., var_m)\, \mathbf{where}\, Sig\, \mathbf{and}\, \mathcal{E}\{s\, \mathbf{return}\, y\} = \mathbf{dispatch}\,(C, M) \\ L' = \mathbf{initframe}\,(M, \mathbf{this} \mapsto v, var_1 \mapsto v'_1, .., var_m \mapsto v'_m) \end{array}}{\ell; g \vdash \langle var = x.M(x_1, .., x_n)\rangle/L/\mu \longrightarrow \mathbf{cx}\,[\ell']\{\,\mathbf{call}\,[M', var, x, x_1, .., x_n, L']\{\langle s\rangle\}\}/L/\mu}$$

ESTEP-CXCAST
$$\frac{pc \Rrightarrow pc' \vdash \ell \Rrightarrow \ell' \quad pc \Rrightarrow pc' \vdash g \Rrightarrow g'}{\ell; g \vdash \langle (pc \Rrightarrow pc')\{s\}\rangle/L/\mu \longrightarrow \mathbf{cx}[pc/\ell/g \Rrightarrow pc'/\ell'/g']\{\langle s\rangle\}/L/\mu}$$

ESTEP-SEQ-DONE
$$\frac{}{\ell; g \vdash \langle \mathbf{done}\rangle; s/L/\mu \longrightarrow \langle s\rangle/L/\mu}$$

ESTEP-CASTCX-DONE
$$\frac{}{\ell; g \vdash \mathbf{cx}[pc/\ell/g \Rrightarrow pc'/\ell'/g']\{\langle \mathbf{done}\rangle\}/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L/\mu}$$

ESTEP-CALLING-DONE
$$\frac{\begin{array}{c} M'(var_1, .., var_m)\, \mathbf{where}\, Sig\, \mathbf{and}\, \mathcal{E}\{s\, \mathbf{return}\, y\} = \mathbf{definition}\,(M) \\ rv[\varsigma] = L'[y] \qquad \varsigma' := \varsigma \sqcup \ell \qquad L' = L[var \mapsto rv[\varsigma']] \qquad \ell \vdash var/L \Downarrow \mathbf{ok} \end{array}}{\ell; g \vdash \mathbf{call}\,[M, var, x, var'_1, .., var'_n, L'']\{\langle \mathbf{done}\rangle\}/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L'/\mu}$$

ESTEP-SEQ-STEP
$$\frac{\ell; g \vdash E(s)/L/\mu \longrightarrow E'(s'_1)/L'/\mu'}{\ell; g \vdash E(s); s_2/L/\mu \longrightarrow E'(s'_1); s_2/L'/\mu'}$$

ESTEP-CX-STEP
$$\frac{g' := g \sqcup \ell' \qquad \ell'; g' \vdash E(s)/L/\mu \longrightarrow E'(s')/L'/\mu'}{\ell; g \vdash \mathbf{cx}\,[\ell']\{E(s)\}/L/\mu \longrightarrow \mathbf{cx}\,[\ell']\{E'(s')\}/L'/\mu'}$$

ESTEP-CALLING-STEP
$$\frac{\bullet; g \vdash E(s)/L'/\mu \longrightarrow E'(s')/L''/\mu'}{\ell; g \vdash \mathbf{call}\,[M', var, x, x_1, .., x_n, L']\{E(s)\}/L/\mu \longrightarrow \mathbf{call}\,[M', var, x, x_1, .., x_n, L'']\{E'(s')\}/L/\mu'}$$

ESTEP-CASTINGCX-STEP
$$\frac{\ell'; g' \vdash E(s)/L/\mu \longrightarrow E'(s')/L'/\mu'}{\ell; g \vdash \mathbf{cx}\,[pc/\ell/g \Rrightarrow pc'/\ell'/g']\{E(s)\}/L/\mu \longrightarrow \mathbf{cx}\,[pc/\ell/g \Rrightarrow pc'/\ell'/g']\{E'(s')\}/L'/\mu'}$$

Fig. 17. Dynamics: complete rules, part 1

$\boxed{\ell; g \vdash rs/L/\mu \longrightarrow rs'/L'/\mu'}$

ESTEP-WHILE-TRUE
$$\frac{rv[\varsigma_1] = L[x] \qquad rv[\varsigma_2] = L[y] \qquad \ell'' := \ell \sqcup \varsigma_1 \sqcup \varsigma_2}{\ell; g \vdash \langle \mathbf{while}\,(x == y)\{s\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\ell'']\{\langle s\rangle\}; \mathbf{while}\,(x == y)\{s\}/L'/\mu'}$$

ESTEP-WHILE-FALSE
$$\frac{rv[\varsigma_1] = L[x] \qquad rv'[\varsigma_2] = L[y] \qquad rv \neq rv'}{\ell; g \vdash \langle \mathbf{while}\,(x == y)\{s\}\rangle/L/\mu \longrightarrow \langle \mathbf{done}\rangle/L/\mu}$$

ESTEP-IF-TRUE
$$\frac{rv[\varsigma_1] = L[x] \qquad rv[\varsigma_2] = L[y] \qquad \varsigma' := \varsigma_1 \sqcup \varsigma_2 \qquad \ell'' := \ell \sqcup \varsigma'}{\ell; g \vdash \langle \mathbf{if}\,(x == y)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\ell'']\{\langle s_1\rangle\}/L/\mu}$$

ESTEP-IF-FALSE
$$\frac{rv[\varsigma_1] = L[x] \qquad rv'[\varsigma_2] = L[y] \qquad rv \neq rv' \qquad \ell'' := \ell \sqcup \varsigma_1 \sqcup \varsigma_2}{\ell; g \vdash \langle \mathbf{if}\,(x == y)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\ell'']\{\langle s_2\rangle\}/L/\mu}$$

ESTEP-IFLABEL-TRUE
$$\frac{rv[\mathbf{D}(A')] = L[x] \qquad A' \sqsubseteq A \qquad L[var] = \mathbf{null} \qquad L' = L[var \mapsto rv[\mathbf{S}]]}{\mathbf{S}; g \vdash \langle \mathbf{iflabel}\,(x \sqsubseteq A; var)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\mathbf{S}]\{\langle s_1\rangle\}/L/\mu}$$

ESTEP-IFLABEL-FALSE
$$\frac{rv[\mathbf{D}(A')] = L[x] \qquad A' \not\sqsubseteq A}{\mathbf{S}; g \vdash \langle \mathbf{iflabel}\,(x \sqsubseteq A; var)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\mathbf{S}]\{\langle s_2\rangle\}/L/\mu}$$

ESTEP-IFLABEL-DYN-TRUE
$$\frac{rv[\mathbf{D}(A')] = L[x] \qquad A' \sqsubseteq A \qquad L[var] = \mathbf{null} \qquad L' = L[var \mapsto rv[\mathbf{S}]]}{\mathbf{D}(A''); g \vdash \langle \mathbf{iflabel}\,(x \sqsubseteq A; var)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\mathbf{D}(A'' \sqcup A)]\{\langle s_1\rangle\}/L/\mu}$$

ESTEP-IFLABEL-DYN-FALSE
$$\frac{rv[\mathbf{D}(A')] = L[x] \qquad A' \not\sqsubseteq A}{\mathbf{D}(A''); g \vdash \langle \mathbf{iflabel}\,(x \sqsubseteq A; var)\{s_1\}\{s_2\}\rangle/L/\mu \longrightarrow \mathbf{cx}[\mathbf{D}(A'' \sqcup A)]\{\langle s_2\rangle\}/L/\mu}$$

Fig. 18. Dynamics: complete rules, part 2

$\boxed{\ell \vdash e/L/\mu \Downarrow v/\mu'}$

STEP-UPD-CONST-STATIC
$$\frac{\ell \in \{\bullet, \mathbf{S}\} \qquad v' = N[\mathbf{S}]}{\ell \vdash N/L/\mu \Downarrow v'/\mu}$$

STEP-UPD-CONST-DYNAMIC
$$\frac{\ell \notin \{\bullet, \mathbf{S}\} \qquad v' = N[\mathbf{D}(PC)]}{\mathbf{D}(PC) \vdash N/L/\mu \Downarrow v'/\mu}$$

STEP-UPD-MOVE
$$\frac{rv[\varsigma] = L[x] \qquad \varsigma' := \varsigma \sqcup \ell}{\ell \vdash x/L/\mu \Downarrow rv[\varsigma']/\mu}$$

STEP-UPD-PLUS
$$\frac{N[\varsigma_1] = L[x] \qquad N'[\varsigma_2] = L[y] \qquad \varsigma' := \varsigma_1 \sqcup \varsigma_2 \qquad \varsigma := \varsigma' \sqcup \ell}{\ell \vdash x + y/L/\mu \Downarrow N + N'[\varsigma]/\mu}$$

STEP-UPD-GETFIELD
$$\frac{oref[\varsigma_1] = L[x] \qquad Obj = \mu[oref] \qquad rv[\varsigma_2] = \mathbf{getfield}\,(F, Obj) \qquad \varsigma := \varsigma_1 \sqcup \varsigma_2 \sqcup \ell}{\ell \vdash x.F/L/\mu \Downarrow rv[\varsigma]/\mu}$$

STEP-UPD-CAST
$$\frac{rv[\varsigma] = L[x] \qquad a' \Rrightarrow a \vdash \varsigma \Rrightarrow \varsigma' \qquad \varsigma'' := \varsigma' \sqcup \ell}{\ell \vdash (a \Leftarrow a')x/L/\mu \Downarrow rv[\varsigma'']/\mu}$$

Fig. 19. Dynamics: evaluation of expressions

$\boxed{a \Rrightarrow a' \vdash \varsigma \Rrightarrow \varsigma'}$

CASTCONV-TRIVIAL
$$\frac{}{a \Rrightarrow a \vdash \varsigma \Rrightarrow \varsigma}$$

CASTCONV-STATIC-TO-DYN
$$\frac{}{A \Rrightarrow \star \vdash \varsigma \Rrightarrow \mathbf{D}(A)}$$

CASTCONV-DYN-TO-STATIC
$$\frac{A \sqsubseteq A'}{\star \Rrightarrow A' \vdash \mathbf{D}(A) \Rrightarrow \mathbf{S}}$$

CASTCONV-PUBLIC-TO-STATIC
$$\frac{}{\bullet \Rrightarrow A \vdash \varsigma \Rrightarrow \mathbf{S}}$$

CASTCONV-PUBLIC-TO-DYN
$$\frac{}{\bullet \Rrightarrow \star \vdash \varsigma \Rrightarrow \mathbf{D}(\bot)}$$

CASTCONV-STATIC-TO-PUBLIC
$$\frac{}{\bot \Rrightarrow \bullet \vdash \mathbf{S} \Rrightarrow \bullet}$$

CASTCONV-DYN-TO-PUBLIC
$$\frac{}{\star \Rrightarrow \bullet \vdash \mathbf{D}(\bot) \Rrightarrow \bullet}$$

Fig. 20. Label conversion

$\boxed{\ell \vdash \mathit{var}/L \Downarrow \mathbf{ok}}$

STEPCHECK-LOCAL-STATIC
$$\frac{\ell \in \{\bullet, \mathbf{S}\}}{\ell \vdash \mathit{var}/L \Downarrow \mathbf{ok}}$$

STEPCHECK-LOCAL-NULL
$$\frac{L[\mathit{var}] = \mathbf{null}}{\mathbf{D}(PC) \vdash \mathit{var}/L \Downarrow \mathbf{ok}}$$

STEPCHECK-LOCAL-DYN-OK
$$\frac{L[\mathit{var}] = \mathit{rv}'[\mathbf{D}(A)] \qquad PC \sqsubseteq A}{\mathbf{D}(PC) \vdash \mathit{var}/L \Downarrow \mathbf{ok}}$$

$\boxed{g \vdash \mathit{oref}[\varsigma]/F/\mu \Downarrow \mathbf{ok}}$

STEPCHECK-GLOBAL-STATIC
$$\frac{}{\mathbf{S} \vdash \mathit{oref}[\varsigma]/F/\mu \Downarrow \mathbf{ok}}$$

STEPCHECK-GLOBAL-DYN-OK
$$\frac{\begin{array}{c}\mu[\mathit{oref}] = \mathit{Obj} \\ \mathit{rv}'[\mathbf{D}(A)] = \mathbf{getfield}\,(F, \mathit{Obj}) \\ PC \sqcup A' \sqsubseteq A\end{array}}{\mathbf{D}(PC) \vdash \mathit{oref}[\mathbf{D}(A'')]/F/\mu \Downarrow \mathbf{ok}}$$

Fig. 21. Dynamics: run-time checks

$\boxed{\Gamma, \alpha \vdash e : \mathcal{C}}$

RT-MOVE
$$\frac{}{\Gamma, \alpha \vdash x : \{\Gamma(x) \leq \alpha\}}$$

RT-PLUS
$$\frac{}{\Gamma, \alpha \vdash x + y : \{\Gamma(x) \leq \alpha, \Gamma(y) \leq \alpha\}}$$

RT-GETFIELD
$$\frac{}{\Gamma, \alpha \vdash x.F : \{\Gamma(x) \leq \alpha, \mathbf{fsec}\,(F) \leq \alpha\}}$$

RT-CONST-STATIC
$$\frac{}{\Gamma, \alpha \vdash N : \{\bot \sim \alpha\}}$$

RT-CONST-DYN
$$\frac{}{\Gamma, \alpha \vdash N_\star[A] : \{\star \sim \alpha\}}$$

RT-CAST
$$\frac{a \lesssim a'}{\Gamma, \alpha \vdash (a' \Leftarrow a)x : \{\Gamma(x) \leq a, a' \leq \alpha\}}$$

Fig. 22. Statics: expression typing

$\boxed{\gamma \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$

ST-LOCAL

$$\frac{\Gamma, \alpha^\dagger \vdash e : \mathcal{C} \quad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger]}{\gamma \vdash var = e : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C} \cup \{\gamma \leq \alpha^\dagger, \gamma \rightarrow_\star \Gamma_1(var)\}, \emptyset}$$

ST-NEW

$$\frac{a_1 .. a_n = \textbf{fieldsecs}\,(C) \quad \mathcal{C} = \{\Gamma_1(x_1) \leq a_1, .., \Gamma_1(x_n) \leq a_n, A \leq \alpha^\dagger\} \quad \mathcal{C}' = \mathcal{C} \cup \{\gamma \leq \alpha^\dagger, \gamma \rightarrow_\star \Gamma_1(var)\} \quad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger]}{\gamma \vdash var = \textbf{new}\, C[A](x_1, .., x_n) : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \{A\}}$$

ST-NEW-DYN

$$\frac{a_1 .. a_n = \textbf{fieldsecs}\,(C) \quad \mathcal{C} = \{\Gamma_1(x_1) \leq a_1, .., \Gamma_1(x_n) \leq a_n\} \quad \mathcal{C}' = \{\star \sim \alpha^\dagger, \gamma \leq \alpha^\dagger, \gamma \rightarrow_\star \Gamma_1(var)\} \quad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger]}{\gamma \vdash var = \textbf{new}_\star\, C[A](x_1, .., x_n) : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \{\star\}}$$

ST-DONE

$$\frac{}{\gamma \vdash \textbf{done} : \Gamma \Rightarrow \Gamma, \emptyset, \emptyset}$$

ST-PUTFIELD

$$\frac{\mathcal{C} = \{\Gamma(x) \leq \textbf{fsec}\,(F), \Gamma(y) \leq \textbf{fsec}\,(F), \gamma \leq \textbf{fsec}\,(F)\}}{\gamma \vdash x.F = y : \Gamma \Rightarrow \Gamma, \mathcal{C}, \{\textbf{fsec}\,(F)\}}$$

ST-CALL

$$\frac{\begin{array}{c} Sig = \textbf{signature}\,(M) \quad \mathcal{E} = \textbf{effects}\,(M) \quad var_1, .., var_n = \textbf{params}\,(M) \\ \mathcal{C}' = Sig[@\textbf{this} \mapsto \Gamma_1(x), @var_1 \mapsto \Gamma_1(x_1), .., @var_n \mapsto \Gamma_1(x_n), @\,\textbf{return} \mapsto \alpha^\dagger] \\ \mathcal{C} = \mathcal{C}' \cup \{\Gamma_1(x) \leq \alpha^\dagger, \gamma \leq \alpha^\dagger, \gamma \rightarrow_\star \Gamma_1(var)\} \cup \bigsqcap_\gamma \mathcal{E} \quad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger] \end{array}}{\gamma \vdash var = x.M(x_1, .., x_n) : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$$

ST-WHILE

$$\frac{\beta^\dagger \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \mathcal{E} \quad \Gamma_3, \mathcal{C}'' = \Gamma_1 \sqcup \Gamma_2 \quad \mathcal{C} = \{\gamma \leq \beta^\dagger, \Gamma_1(x) \leq \beta^\dagger, \Gamma_1(y) \leq \beta^\dagger\} \cup \mathcal{C}'}{\gamma \vdash \textbf{while}\,(x == y)\{s\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}}$$

ST-IF

$$\frac{\beta^\dagger \vdash s_1 : \Gamma_1 \Rightarrow \Gamma_2', \mathcal{C}_1, \mathcal{E}_1 \quad \beta^\dagger \vdash s_2 : \Gamma_1 \Rightarrow \Gamma_2'', \mathcal{C}_2, \mathcal{E}_2 \quad \mathcal{C}' = \{\gamma \leq \beta^\dagger, \Gamma_1(x) \leq \beta^\dagger, \Gamma_1(y) \leq \beta^\dagger\} \quad \Gamma_2, \mathcal{C}'' = \Gamma_2' \sqcup \Gamma_2'' \quad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}' \cup \mathcal{C}''}{\gamma \vdash \textbf{if}\,(x == y)\{s_1\}\{s_2\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}_1 \cup \mathcal{E}_2}$$

ST-IFLABEL

$$\frac{\Gamma_1' = \Gamma_1[var \mapsto \alpha^\dagger] \quad \beta^\dagger \vdash s_1 : \Gamma_1' \Rightarrow \Gamma_2', \mathcal{C}_1, \mathcal{E}_1 \quad \beta^\dagger \vdash s_2 : \Gamma_1 \Rightarrow \Gamma_2'', \mathcal{C}_2, \mathcal{E}_2 \quad a \in \{A, \star\} \quad \mathcal{C}' = \{\gamma \leq \beta^\dagger, a \leq \beta^\dagger, \Gamma_1(x) \leq \star, A \leq \alpha^\dagger\} \quad \Gamma_2, \mathcal{C}'' = \Gamma_2' \sqcup \Gamma_2'' \quad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}' \cup \mathcal{C}''}{\gamma \vdash \textbf{iflabel}\,(x \sqsubseteq A; var)\{s_1\}\{s_2\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}_1 \cup \mathcal{E}_2}$$

ST-IFPC

$$\frac{\beta^\dagger \vdash s_1 : \Gamma_1 \Rightarrow \Gamma_2', \mathcal{C}_1, \mathcal{E}_1 \quad \gamma \vdash s_2 : \Gamma_1 \Rightarrow \Gamma_2'', \mathcal{C}_2, \mathcal{E}_2 \quad \mathcal{C}' = \{\gamma \leq \star, A \leq \beta^\dagger\} \quad \Gamma_2, \mathcal{C}'' = \Gamma_2' \sqcup \Gamma_2'' \quad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}' \cup \mathcal{C}''}{\gamma \vdash \textbf{ifpc}\,(A)\{s_1\}\{s_2\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E}_1 \cup \mathcal{E}_2}$$

ST-SEQ

$$\frac{\gamma \vdash s_1 : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}_1, \mathcal{E}_1 \quad \gamma \vdash s_2 : \Gamma_2 \Rightarrow \Gamma_3, \mathcal{C}_2, \mathcal{E}_2}{\gamma \vdash s_1; s_2 : \Gamma_1 \Rightarrow \Gamma_3, \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{E}_1 \cup \mathcal{E}_2}$$

ST-CXCAST

$$\frac{\beta^\dagger \vdash s : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \mathcal{E} \quad \mathcal{C} = \{pc \leq \beta^\dagger, \gamma \leq pc'\} \cup \mathcal{C}' \quad pc' \lesssim pc}{\gamma \vdash (pc' \Rrightarrow pc)\{s\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \{pc'\}}$$

Fig. 23. Statics: statement typing

$\boxed{\gamma, \ell, g \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell', g'}$

ET-HOLE

$$\frac{}{\gamma, \ell, g \vdash \langle\rangle : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell, g}$$

ET-CX

$$\frac{\begin{array}{c}\ell \leq \ell' \qquad g \leq g' \qquad \ell' \leq g' \\ \beta^\dagger, \ell', g' \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \mathcal{E} | \ell'', g'' \\ \mathcal{C} = \mathcal{C}' \cup \{\gamma \leq \beta^\dagger\}\end{array}}{\gamma, \ell, g \vdash \mathbf{cx}\,[\ell']\{E\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell'', g''}$$

ET-CALLING

$$\frac{\begin{array}{c}var_1, \ldots, var_n = \mathbf{params}\,(M) \\ Sig = \mathbf{signature}\,(M) \\ y = \mathbf{return}\,(M) \\ \mathcal{E} = \mathbf{effects}\,(M) \qquad \Gamma_1', \mathcal{C}' \vdash L' \\ \bullet, \bullet, g \vdash E : \Gamma_1' \Rightarrow \Gamma_2', \mathcal{C}', \mathcal{E}' | \ell'', g'' \\ \mathcal{C}' \cup \{\Gamma_2'(y) \leq \mathbf{return}\} <: \mathbf{constraints}\,(Sig) \\ \mathcal{E}' <: \mathcal{E} \qquad \Gamma_2 = \Gamma_1[var \mapsto \alpha^\dagger] \\ \mathcal{C}'' = \{\Gamma_1(x) \leq \alpha^\dagger\} \cup Sig[@var_1 \mapsto \Gamma_1(x_1'), \ldots, @var_n \mapsto \Gamma_1(x_n'), @\,\mathbf{return} \mapsto \alpha^\dagger] \\ \mathcal{C} = \mathcal{C}'' \cup \prod_\gamma \mathcal{E}\end{array}}{\gamma, \ell, g \vdash \mathbf{call}\,[M, var, x, x_1', \ldots, x_n', L']\{E\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}, \mathcal{E} | \ell'', g''}$$

ET-CXCAST

$$\frac{\begin{array}{c}pc \lesssim pc' \qquad pc \Rrightarrow pc' \vdash \ell \Rrightarrow \ell' \\ pc \Rrightarrow pc' \vdash g \Rrightarrow g' \\ \beta^\dagger, \ell', g' \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}', \mathcal{E}' | \ell'', g'' \\ \mathcal{C}'' = \{\gamma \leq pc, pc' \leq \beta^\dagger\}\end{array}}{\gamma, \ell, g \vdash \mathbf{cx}\,[pc/\ell/g \Rrightarrow pc'/\ell'/g']\{E\} : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}' \cup \mathcal{C}'', \{pc\} | \ell'', g''}$$

ET-SEQ

$$\frac{\begin{array}{c}\gamma, \ell, g \vdash E : \Gamma_1 \Rightarrow \Gamma_2, \mathcal{C}_1, \mathcal{E}_1 | \ell'', g'' \\ \gamma \vdash s_2 : \Gamma_2 \Rightarrow \Gamma_3, \mathcal{C}_2, \mathcal{E}_2\end{array}}{\gamma, \ell, g \vdash E; s_2 : \Gamma_1 \Rightarrow \Gamma_3, \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{E}_1 \cup \mathcal{E}_2 | \ell'', g''}$$

Fig. 24. Statics: typing of execution contexts

$\boxed{a \lesssim a'}$

CSTBL-FROM-DYNAMIC

$$\frac{}{\star \lesssim a}$$

CSTBL-TO-DYNAMIC

$$\frac{}{a \lesssim \star}$$

CSTBL-PUBLIC-TO-STATIC

$$\frac{}{\bullet \lesssim A}$$

CSTBL-STATIC-TO-PUBLIC

$$\frac{}{\bot \lesssim \bullet}$$

Fig. 25. Statics: typing of execution contexts